

OS Concepts

By Jani Ahamed Habeeb Mohamed

Table of Contents

1. Basics	9
1.1 The boot process	10
BIOS POST	10
GRUB2	11
1.1.1.1 Stage 1	12
1.1.1.2 Stage 1.5	12
1.1.1.3 Stage 2	13
Kernel	13
1.2 The startup process.....	14
systemd	14
1.3 Services Provided by System Calls.....	19
2. CPU Scheduling	21
What does a process look like in memory?	21
Explanation of Process.....	22
Attributes or Characteristics of a Process	22
States of Process	22
When Does Context Switching Happen?.....	23
Context Switch vs Mode Switch	23
CPU-Bound vs I/O-Bound Processes	23
Scheduling Algorithms.....	23
Process States in Operating System.....	24
2.1 Types of Schedulers.....	26
Multiprogramming	26
Degree of Multiprogramming	27
Operation on the Process	27
2.2 Features of the Process State.....	27
2.3 Additional Points to Consider for Process Control Block (PCB).....	30
2.4 Advantages-	30
2.5 Disadvantages-	31
2.6 Categories in Scheduling.....	32
2.7 Long Term or Job Scheduler	32
2.8 Short-Term or CPU Scheduler	32
2.9 Medium-Term Scheduler	33
2.10 Some Other Schedulers	33
2.11 Context Switching	33
2.12 What is a process?.....	34
2.13 How is process memory used for efficient operation?	34
2.14 What is Process Scheduling?	35
2.15 Why do we need to schedule processes?	35
2.16 What is the need for CPU scheduling algorithm?	36
Objectives of Process Scheduling Algorithm:	36
2.17 What are the different terminologies to take care of in any CPU Scheduling algorithm?	37

2.18	Things to take care while designing a CPU Scheduling algorithm?	37
2.19	What are the different types of CPU Scheduling Algorithms?	37
1.	First Come First Serve:	38
2.	Shortest Job First(SJF):	38
3.	Longest Job First(LJF):	39
4.	Priority Scheduling:.....	40
5.	Round robin:	41
6.	Shortest Remaining Time First:.....	41
7.	Longest Remaining Time First:.....	41
8.	Highest Response Ratio Next:.....	42
9.	Multiple Queue Scheduling:	43
10.	Multilevel Feedback Queue Scheduling:.....	43
2.20	Comparison between various CPU Scheduling algorithms	44
2.21	Preemptive Scheduling	47
2.22	Non-Preemptive Scheduling	48
2.23	Key Differences Between Preemptive and Non-Preemptive Scheduling.....	49
	Comparison Chart	50
	Features of Belady's Anomaly:	60
	Advantages :	61
	Program for Non-Preemptive Shortest Job First CPU Scheduling.....	63
	Examples to show working of Non-Preemptive Shortest Job First CPU Scheduling Algorithm:	63
2.24	Characteristics of Round Robin CPU Scheduling Algorithm	76
2.25	Advantages of Round Robin CPU Scheduling Algorithm.....	77
2.26	Disadvantages of Round Robin CPU Scheduling Algorithm	77
2.27	Examples to show working of Round Robin Scheduling Algorithm.....	77
	How to compute below times in Round Robin using a program?	81
	Advantages of HRRN CPU Scheduling.....	87
	Disadvantages of HRRN CPU Scheduling	87
	Performance of HRRN –	87
	Features of Multilevel Queue (MLQ) CPU Scheduling:	94
2.28	Advantages of Multilevel Queue CPU Scheduling:	94
2.29	Disadvantages of Multilevel Queue CPU Scheduling:.....	95
	Features of Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling:	97
	What is the need for such complex Scheduling?	99
	Advantages of Lottery Process Scheduling:	100
	Disadvantages of Lottery Process Scheduling:.....	101
	7. Approaches to Multiple-Processor Scheduling –	101
	Processor Affinity –	101
	Load Balancing –	102
	Multicore Processors –	102
	Virtualization and Threading –	103
3.	Process and Threads.....	105
3.1	Why Do We Need Thread?.....	105
3.2	Why Multi-Threading?.....	105
3.3	Difference Between Process and Thread	105

3.4 Advantages of Thread.....	106
3.5 Types of Threads	106
User Level Threads	107
Kernel Level Threads	107
3.6 Components of Threads.....	108
Different ways in which the creation of Zombie can be Prevented	113
4. Disk Management	122
The advantages of using a file system.....	122
Disadvantages of using a file system	122
4.1 The Following Issues Are Handled By The File Syst4em	122
4.2 FILE ALLOCATION METHODS	123
Continuous Allocation	123
Linked Allocation(Non-contiguous allocation).....	125
Indexed Allocation	125
Disk Free Space Management	126
Here are some key differences between privileged and non-privileged instructions:	132
Acyclic Graph Structure:	134
Primary Memory	139
Secondary Memory	139
Use of Secondary memory	139
Types of Secondary memory	140
4.3 What is a Hard Disk Drive?.....	140
4.4 How do Hard Disk Drives Work?	142
4.5 Hard Disk Drive Storage Capacity.....	142
4.6 History of HDD	142
4.7 Features of HDD	143
4.7.1.1 Need of Spooling.....	145
4.7.1.2 How Does Spooling Work:	146
4.8 Key Terms Associated with Disk Scheduling	148
4.9 Disk Scheduling Algorithms.....	149
FCFS (First Come First Serve).....	149
SSTF (Shortest Seek Time First)	150
SCAN	150
C-SCAN	151
4.9.1.1 Example:.....	152
LOOK	152
4.9.1.2 Example:.....	153
C-LOOK	153
4.9.1.3 Example:.....	153
RSS (Random Scheduling)	154
LIFO (Last-In First-Out)	154
N-STEP SCAN.....	154
F-SCAN	155
5. Memory Management	156
5.1 Types of Memory Hierarchy.....	156
5.2 Memory Hierarchy Design	156

1. Registers	156
2. Cache Memory	156
3. Main Memory	157
5.2.1.1 Types of Main Memory.....	157
4. Secondary Storage	157
5.3 System-Supported Memory Standards.....	157
5.4 Types of Computer Memory	158
5.5 Units of Memory	159
1. Bit.....	159
2. Nibble	160
3. Word	160
4. Bytes.....	160
5. Kilobyte.....	160
6. Megabyte.....	160
7. Gigabyte	160
8. Terabyte	161
9. Petabyte	161
10. Exabyte (1024 petabytes).....	161
11. Zettabyte (1024 exabytes).....	161
12. Yottabyte	161
5.6 Conversations of units	162
5.7 What is SRAM?	162
SRAM Memory Cell	162
5.8 What is DRAM?	163
DRAM Memory Cell	163
5.9 Types of DRAM.....	164
5.10 Difference Between SRAM and DRAM	165
Here are the basic steps in the Buddy System memory allocation technique:.....	166
Difference between Internal fragmentation and External fragmentation	169
5.11 Types of Memory Management Techniques.....	175
5.12 Variable (or Dynamic) Partitioning.....	175
5.13 Advantages of Variable Partitioning.....	176
5.14 Disadvantages of Variable Partitioning	176
5.15 Key Points On Variable (or Dynamic) Partitioning in Operating Systems	177
• There are several advantages to non-contiguous allocation.	178
• However, non-contiguous allocation also has some disadvantages.....	178
Here are some important points about logical and physical addresses in operating systems:	179
5.16 Terminologies Associated with Memory Control.....	187
5.17 Why Multi-Level Paging is Required?	189
5.18 What is an Inverted Page Table?.....	191
5.19 Components of Inverted Page Table	191
5.20 Types of Segmentation in Operating System	192
5.21 What is Segment Table?	192
5.22 Advantages of Segmentation in Operating System.....	194

5.23	Disadvantages of Segmentation in Operating System.....	195
5.24	Information Stored in Page Table Entry	204
5.25	Advantages of Using a Page Table in a Virtual Memory System	205
5.26	Demand Paging	206
5.27	Advantages of Virtual Memory	207
5.28	Disadvantages of Virtual Memory.....	207
5.29	Swapping	208
5.30	Thrashing	208
5.31	Causes of Thrashing.....	209
5.32	Recovery of Thrashing	209
5.33	Performance in Virtual Memory	209
5.34	Frame Allocation	209
	Paging Policies	210
	Page Table in DBMS:	213
	Inverted Page Table in DBMS:	213
5.35	Types of Segmentation in Operating System	214
5.36	What is Segment Table?	215
	Steps in TLB hit.....	217
	Steps in TLB miss.....	217
5.37	What is Next Fit?	217
5.38	What is its advantage over first fit?	218
	Page Replacement Algorithms:	220
	Complexity Analysis :.....	228
	1. Buddy system –	235
	2. Slab Allocation –	238
6.	Process Synchronization	245
6.1	Race Condition	245
	Actual meaning of race-condition	246
6.2	Critical Section Problem	246
6.3	Peterson's Solution	247
6.4	Disadvantages of Peterson's Solution	248
6.5	Semaphores	248
6.6	Advantages of Process Synchronization	249
6.7	Disadvantages of Process Synchronization	249
6.8	Critical Section	250
	Critical Section Problem.....	250
	Advantages of critical section in process synchronization:.....	252
	Disadvantages of critical section in process synchronization:	253
6.9	Important points related to critical section in process synchronization are:	253
6.10	Used System Calls.....	261
6.11	Shared Memory for Writer Process	262

6.12	CPP	262
6.13	Shared Memory for Reader Process	263
6.14	CPP	263
6.15	C	268
6.16	Mutex	280
	Advantages of Mutex	281
	Disadvantages of Mutex	281
	Using Mutex.....	281
6.17	Semaphore.....	281
	Advantages of Semaphore	281
	Disadvantages of Semaphore.....	282
	6.17.1.1 Using Semaphore	282
6.18	Difference Between Mutex and Semaphore.....	282
6.19	Misconception.....	283
	6.19.1.1 Explanation:	290
6.20	C	290
	6.20.1.1 Wastage of CPU clock cycles	292
	6.20.1.2 Memory fence.....	293
7.	Program on Dining Philosopher Problem.....	310
	Solution to Dining Philosopher Problem	310
	How it works?	313
	Output	314
7.1	Deadlock Characteristics.....	328
7.2	Deadlock Prevention	328
7.3	Deadlock Avoidance	329
	Resource Allocation Graph	329
	Banker's Algorithm.....	329
	7.3.1.1 Inputs to Banker's Algorithm	330
	7.3.1.2 The request will only be granted under the below condition	330
	There are two main approaches to deadlock detection and recovery:	331
	ADVANTAGES OR DISADVANTAGES:	333
	Advantages of Deadlock Detection and Recovery in Operating Systems:	333
	Disadvantages of Deadlock Detection and Recovery in Operating Systems:	333
	What leads to Livelocks?	336
	Difference between Deadlock, Starvation, and Livelock:	336
7.4	Why Banker's Algorithm is Named So?	338
	Available.....	339
	Max	339
	Allocation	339
	Need.....	339
	Banker's Algorithm.....	339
7.5	Safety Algorithm	340
	Q.1: What will be the content of the Need matrix?	341
	Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?	342
	Q.3: What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?	342

7.6 Resource Allocation Graph (RAG).....	346
Types of Vertices in RAG.....	346
How many Types of Edges are there in RAG?.....	347
Example 1 (Single instances RAG).....	348

1. Basics

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently. An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, a memory management module, I/O programs, and a file system.

IO management, File System management, Job schedulers, Memory management, Processor management, security, error detection, throughput.

- **Resource Management:** When parallel accessing happens in the OS means when multiple users are accessing the system the OS works as Resource Manager, Its responsibility is to provide hardware to the user. It decreases the load in the system.
- **Process Management:** It includes various tasks like **scheduling and termination** of the process. It is done with the help of [CPU Scheduling algorithms](#).
- **Storage Management:** The **file system** mechanism used for the management of the storage. [NIFS](#), [CIFS](#), [CFS](#), [NFS](#), etc. are some file systems. All the data is stored in various tracks of Hard disks that are all managed by the storage manager. It included **Hard Disk**.
- **Memory Management:** Refers to the management of primary memory. The operating system has to keep track of how much memory has been used and by whom. It has to decide which process needs memory space and how much. OS also has to allocate and deallocate the memory space.
- **Security/Privacy Management:** Privacy is also provided by the Operating system using passwords so that unauthorized applications can't access programs or data. For example, Windows uses **Kerberos** authentication to prevent unauthorized access to data.

- [Batch Operating System](#)
- [Multi-Programming System](#)
- Multi-Processing System
- Multi-Tasking Operating System
- [Time-Sharing Operating System](#)
- Distributed Operating System
- [Network Operating System](#)
- [Real-Time Operating System](#)

DRAM	SRAM
1. Constructed of tiny capacitors that leak electricity.	1. Constructed of circuits similar to D flip-flops.
2. Requires a recharge every few milliseconds to maintain its data.	2. Holds its contents as long as power is available.
3. Inexpensive.	3. Expensive.
4. Slower than SRAM.	4. Faster than DRAM.
5. Can store many bits per chip.	5. Can not store many bits per chip.
6. Uses less power.	6. Uses more power.
7. Generates less heat.	7. Generates more heat.
8. Used for main memory.	8. Used for cache.

Difference between SRAM and DRAM

Linux boot sequence:

- BIOS POST
- Boot loader (GRUB2)
- Kernel initialization
- Start systemd, the parent of all processes.

Note that this article covers GRUB2 and systemd because they are the current boot loader and initialization software for most major distributions. Other software options have been used historically and are still found in some distributions.

1.1 The boot process

The boot process can be initiated in one of a couple ways. First, if power is turned off, turning on the power will begin the boot process. If the computer is already running a local user, including root or an unprivileged user, the user can programmatically initiate the boot sequence by using the GUI or command line to initiate a reboot. A reboot will first do a shutdown and then restart the computer.

BIOS POST

The first step of the Linux boot process really has nothing whatever to do with Linux. This is the hardware portion of the boot process and is the same for any

operating system. When power is first applied to the computer it runs the POST (Power On Self Test) which is part of the BIOS (Basic I/O System).

When IBM designed the first PC back in 1981, BIOS was designed to initialize the hardware components. POST is the part of BIOS whose task is to ensure that the computer hardware functioned correctly. If POST fails, the computer may not be usable and so the boot process does not continue.

BIOS POST checks the basic operability of the hardware and then it issues a BIOS [interrupt](#), INT 13H, which locates the boot sectors on any attached bootable devices. The first boot sector it finds that contains a valid boot record is loaded into RAM and control is then transferred to the code that was loaded from the boot sector.

The boot sector is really the first stage of the boot loader. There are three boot loaders used by most Linux distributions, GRUB, GRUB2, and LILO. GRUB2 is the newest and is used much more frequently these days than the other older options.

GRUB2

GRUB2 stands for "GRand Unified Bootloader, version 2" and it is now the primary bootloader for most current Linux distributions. GRUB2 is the program which makes the computer just smart enough to find the operating system kernel and load it into memory. Because it is easier to write and say GRUB than GRUB2, I may use the term GRUB in this document but I will be referring to GRUB2 unless specified otherwise.

GRUB has been designed to be compatible with the [multiboot specification](#) which allows GRUB to boot many versions of Linux and other free operating systems; it can also chain load the boot record of proprietary operating systems.

GRUB can also allow the user to choose to boot from among several different kernels for any given Linux distribution. This affords the ability to boot to a previous kernel version if an updated one fails somehow or is incompatible with an important piece of software. GRUB can be configured using the /boot/grub/grub.conf file.

GRUB1 is now considered to be legacy and has been replaced in most modern distributions with GRUB2, which is a rewrite of GRUB1. Red Hat based distros upgraded to GRUB2 around Fedora 15 and CentOS/RHEL 7. GRUB2 provides the same boot functionality as GRUB1 but GRUB2 is also a mainframe-like

command-based pre-OS environment and allows more flexibility during the pre-boot phase. GRUB2 is configured with /boot/grub2/grub.cfg.

The primary function of either GRUB is to get the Linux kernel loaded into memory and running. Both versions of GRUB work essentially the same way and have the same three stages, but I will use GRUB2 for this discussion of how GRUB does its job. The configuration of GRUB or GRUB2 and the use of GRUB2 commands is outside the scope of this article.

Although GRUB2 does not officially use the stage notation for the three stages of GRUB2, it is convenient to refer to them in that way, so I will in this article.

1.1.1.1 Stage 1

As mentioned in the BIOS POST section, at the end of POST, BIOS searches the attached disks for a boot record, usually located in the Master Boot Record (MBR), it loads the first one it finds into memory and then starts execution of the boot record. The bootstrap code, i.e., GRUB2 stage 1, is very small because it must fit into the first 512-byte sector on the hard drive along with the partition table. The total amount of space allocated for the actual bootstrap code in a [classic generic MBR](#) is 446 bytes. The 446 Byte file for stage 1 is named boot.img and does not contain the partition table which is added to the boot record separately.

Because the boot record must be so small, it is also not very smart and does not understand filesystem structures. Therefore the sole purpose of stage 1 is to locate and load stage 1.5. In order to accomplish this, stage 1.5 of GRUB must be located in the space between the boot record itself and the first partition on the drive. After loading GRUB stage 1.5 into RAM, stage 1 turns control over to stage 1.5.

1.1.1.2 Stage 1.5

As mentioned above, stage 1.5 of GRUB must be located in the space between the boot record itself and the first partition on the disk drive. This space was left unused historically for technical reasons. The first partition on the hard drive begins at sector 63 and with the MBR in sector 0, that leaves 62 512-byte sectors—31,744 bytes—in which to store the core.img file which is stage 1.5 of GRUB. The core.img file is 25,389 Bytes so there is plenty of space available between the MBR and the first disk partition in which to store it.

Because of the larger amount of code that can be accommodated for stage 1.5, it can have enough code to contain a few common filesystem drivers, such as the

standard EXT and other Linux filesystems, FAT, and NTFS. The GRUB2 core.img is much more complex and capable than the older GRUB1 stage 1.5. This means that stage 2 of GRUB2 can be located on a standard EXT filesystem but it cannot be located on a logical volume. So the standard location for the stage 2 files is in the /boot filesystem, specifically /boot/grub2.

Note that the /boot directory must be located on a filesystem that is supported by GRUB. Not all filesystems are. The function of stage 1.5 is to begin execution with the filesystem drivers necessary to locate the stage 2 files in the /boot filesystem and load the needed drivers.

1.1.1.3 Stage 2

All of the files for GRUB stage 2 are located in the /boot/grub2 directory and several subdirectories. GRUB2 does not have an image file like stages 1 and 2. Instead, it consists mostly of runtime kernel modules that are loaded as needed from the /boot/grub2/i386-pc directory.

The function of GRUB2 stage 2 is to locate and load a Linux kernel into RAM and turn control of the computer over to the kernel. The kernel and its associated files are located in the /boot directory. The kernel files are identifiable as they are all named starting with vmlinuz. You can list the contents of the /boot directory to see the currently installed kernels on your system.

GRUB2, like GRUB1, supports booting from one of a selection of Linux kernels. The Red Hat package manager, DNF, supports keeping multiple versions of the kernel so that if a problem occurs with the newest one, an older version of the kernel can be booted. By default, GRUB provides a pre-boot menu of the installed kernels, including a rescue option and, if configured, a recovery option.

Stage 2 of GRUB2 loads the selected kernel into memory and turns control of the computer over to the kernel.

Kernel

All of the kernels are in a self-extracting, compressed format to save space. The kernels are located in the /boot directory, along with an initial RAM disk image, and device maps of the hard drives.

After the selected kernel is loaded into memory and begins executing, it must first extract itself from the compressed version of the file before it can perform any useful work. Once the kernel has extracted itself, it loads [systemd](#), which is the replacement for the old [SysV init](#) program, and turns control over to it.

This is the end of the boot process. At this point, the Linux kernel and systemd are running but unable to perform any productive tasks for the end user because nothing else is running.

1.2 The startup process

The startup process follows the boot process and brings the Linux computer up to an operational state in which it is usable for productive work.

systemd

systemd is the mother of all processes and it is responsible for bringing the Linux host up to a state in which productive work can be done. Some of its functions, which are far more extensive than the old init program, are to manage many aspects of a running Linux host, including mounting filesystems, and starting and managing system services required to have a productive Linux host. Any of systemd's tasks that are not related to the startup sequence are outside the scope of this article.

First, systemd mounts the filesystems as defined by **/etc/fstab**, including any swap files or partitions. At this point, it can access the configuration files located in /etc, including its own. It uses its configuration file, **/etc/systemd/system/default.target**, to determine which state or target, into which it should boot the host. The **default.target** file is only a symbolic link to the true target file. For a desktop workstation, this is typically going to be the **graphical.target**, which is equivalent to **runlevel 5** in the old SystemV init. For a server, the default is more likely to be the **multi-user.target** which is like **runlevel 3** in SystemV. The **emergency.target** is similar to single user mode.

Note that targets and services are systemd units.

Table 1, below, is a comparison of the systemd targets with the old SystemV startup runlevels. The **systemd target aliases** are provided by systemd for backward compatibility. The target aliases allow scripts—and many sysadmins like myself—to use SystemV commands like **init 3** to change runlevels. Of course, the SystemV commands are forwarded to systemd for interpretation and execution.

SystemV Runlevel	systemd target	systemd target aliases	Description

	halt.target		Halts the system without powering it down.
0	poweroff.target	runlevel0.target	Halts the system and turns the power off.
S	emergency.target		Single user mode. No services are running; filesystems are mounted read-only. This is the most basic level of operation with only an emergency shell running on the main console for the user to interact with.
1	rescue.target	runlevel1.target	A base system including mounting the filesystems with read/write access, all services running and a rescue shell on the main console.
2		runlevel2.target	Multiuser, without NFS but all other non-GUI services running.
3	multi-user.target	runlevel3.target	All services running but command line interface (CLI) only.
4		runlevel4.target	Unused.
5	graphical.target	runlevel5.target	multi-user with a GUI.
6	reboot.target	runlevel6.target	Reboot
	default.target		This target is always aliased with a symbolic link to either graphical.target or graphical.target. systemd always uses the default.target for the system. The default.target should never be aliased to poweroff.target, or reboot.target.

Table 1: Comparison of SystemV runlevels with systemd targets and some target aliases.

Each target has a set of dependencies described in its configuration file. systemd starts the required dependencies. These dependencies are the services required to run the Linux host at a specific level of functionality. When all of the dependencies listed in the target configuration files are loaded and running, the system is running at that target level.

systemd also looks at the legacy SystemV init directories to see if any startup files exist there. If so, systemd uses those as configuration files to start the services described by the files. The deprecated network service is a good example of one of those that still use SystemV startup files in Fedora.

Figure 1, below, is copied directly from the [bootup man page](#). It shows the general sequence of events during systemd startup and the basic ordering requirements to ensure a successful startup.

The **sysinit.target** and **basic.target** targets can be considered as checkpoints in the startup process. Although systemd has as one of its design goals to start system services in parallel, there are still certain services and functional targets that must be started before other services and targets can be started. These checkpoints cannot be passed until all of the services and targets required by that checkpoint are fulfilled.

So the **sysinit.target** is reached when all of the units on which it depends are completed. All of those units, mounting filesystems, setting up swap files, starting udev, setting the random generator seed, initiating low-level services, and setting up cryptographic services if one or more filesystems are encrypted, must be completed, but within the **sysinit.target** those tasks can be performed in parallel.

The **sysinit.target** starts up all of the low-level services and units required for the system to be marginally functional and that are required to enable moving on to the **basic.target**.

After the **sysinit.target** is fulfilled, systemd next starts the **basic.target**, starting all of the units required to fulfill it. The basic target provides some additional functionality by starting units that are required for the next target. These include setting up things like paths to various executable directories, communication sockets, and timers.

Finally, the user-level targets, **multi-user.target** or **graphical.target** can be initialized. Notice that the **multi-user.target** must be reached before the graphical target dependencies can be met.

The underlined targets in Figure 1, are the usual startup targets. When one of these targets is reached, then startup has completed. If the **multi-user.target** is the default, then you should see a text mode login on the console.

If **graphical.target** is the default, then you should see a graphical login; the specific GUI login screen you see will depend on the default [display manager](#) you use.

- BIOS can boot from drives of less than 2 TB. 3+ TB drives are now standard, and a system with a BIOS can't boot from them.
- BIOS runs in 16-bit processor mode and has only 1 MB of space to execute.
- It can't initialize multiple hardware devices at once, thus leading to a slow booting process.
- BIOS is limited in its configuration options and can be difficult to use for advanced users.
- BIOS is vulnerable to malware attacks, such as rootkits, that can exploit vulnerabilities in the firmware.

UEFI, or Unified Extensible Firmware Interface, is a type of firmware interface used in modern computers to replace the traditional BIOS (Basic Input/Output System). UEFI provides a more advanced and flexible interface than BIOS, allowing for improved boot times, more efficient power management, and support for larger hard drives and partitions.

One of the main differences between UEFI and BIOS is the way they handle system initialization. With BIOS, the firmware is stored in a read-only memory (ROM) chip on the motherboard, and the firmware is responsible for loading the operating system from the hard drive. In contrast, UEFI is stored in non-volatile memory on the motherboard and is capable of directly loading the operating system from the hard drive.

UEFI also provides more advanced features than BIOS, such as secure boot, which helps prevent malicious software from loading during the boot process. UEFI also supports graphical interfaces and provides more detailed system information and configuration options than BIOS.

Difference between the Booting Process with UEFI and the Booting Process with BIOS

- **Booting Process With BIOS :** When BIOS begins its execution, it first goes for the [Power-On Self Test \(POST\)](#), which ensures that the hardware devices are functioning correctly. After that, it checks for the Master Boot Record in the first sector of the selected boot device. From the MBR, the location of the Boot-Loader is retrieved, which, after being loaded by BIOS into the computer's [RAM](#), loads the operating system into the main memory.
- **Booting Process With UEFI :** Unlike BIOS, [UEFI](#) doesn't look for the MBR in the first sector of the Boot Device. It maintains a list of valid boot volumes called EFI System Partitions. During the POST procedure, the UEFI firmware scans all of the bootable storage devices that are connected to the system for a valid GUID Partition Table (GPT), which is an improvement over MBR. Unlike the MBR, GPT doesn't contain a Boot-Loader. The firmware itself scans the GPT to find an EFI System Partition to boot from, and directly loads the OS from the right partition.

If it fails to find one, it goes back to the BIOS-type Booting process called ‘Legacy Boot’.

Advantages of UEFI over BIOS

- **Breaking Out Of Size Limitations :** The UEFI firmware can boot from drives of 2.2 TB or larger with the theoretical upper limit being 9.4 zettabytes, which is roughly 3 times the size of the total information present on the Internet. This is due to the fact that GPT uses 64-bit entries in its table, thereby dramatically expanding the possible boot-device size.
- **Speed and performance :** UEFI can run in 32-bit or 64-bit mode and has more addressable address space than BIOS, which means your boot process is faster.
- **More User-Friendly Interface :** Since UEFI can run in 32-bit and 64-bit mode, it provides a better UI configuration that has better graphics and also supports mouse cursor.
- **Security:** UEFI also provides the feature of Secure Boot. It allows only authentic drivers and services to load at boot time, to make sure that no malware can be loaded at computer startup. It also requires drivers and the Kernel to have a digital signature, which makes it an effective tool in countering piracy and boot-sector malware.
- **Uses GPT instead of MBR:** As mentioned earlier, UEFI firmware scans all of the bootable storage devices connected to the system for a valid GUID Partition Table (GPT), which is an improvement over the Master Boot Record (MBR) used by BIOS. GPT is more advanced and can support larger drives and partitions, as well as more partitions per drive.
- **Handle multiple operating systems:** With UEFI, it’s possible to have multiple operating systems installed on a single hard drive or SSD, and the firmware can manage them all. This is because UEFI maintains a boot manager that can choose which operating system to load at startup.
- **More detailed system information:** UEFI can provide more detailed information about the system hardware and firmware configuration, including information about the CPU, memory, storage devices, and firmware settings. This information can be useful for troubleshooting and diagnostics.
- **Supports network booting:** UEFI firmware can support network booting, which means that the operating system can be loaded from a server on the network rather than from local storage. This can be useful for remote management and deployment of operating systems.
- **Supports more modern technologies:** UEFI supports newer technologies such as USB 3.0, NVMe, and PCIe, which can improve performance and functionality compared to BIOS.

UEFI doesn’t require a Boot-Loader, and can also operate alongside BIOS, supporting legacy boot, which in turn, makes it compatible with older operating systems. Intel plans to completely replace BIOS with UEFI, for all its chipsets, by 2020.

The **getrlimit()** and **setrlimit()** system calls can be used to get and set the resource limits such as files, CPU, memory etc. associated with a process.

A user program can interact with the operating system using a system call. A number of services are requested by the program, and the OS responds by launching a number of systems calls to fulfill the request. A system call can be written in high-level languages like C or Pascal or in assembly language. If a high-level language is used, the operating system may directly invoke system calls, which are predefined functions.

A system call is initiated by the program executing a specific instruction, which triggers a switch to kernel mode, allowing the program to request a service from the OS. The OS then handles the request, performs the necessary operations, and returns the result back to the program.

System calls are essential for the proper functioning of an operating system, as they provide a standardized way for programs to access system resources. Without system calls, each program would need to implement its own methods for accessing hardware and system services, leading to inconsistent and error-prone behavior.

1.3 Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory, and File system management
- Device handling(I/O)
- Protection
- Networking, etc.
 - **Process control:** end, abort, create, terminate, allocate, and free memory.
 - **File management:** create, open, close, delete, read files,s, etc.
 - **Device management**
 - **Information maintenance**
 - **Communication**

Process	Windows	Quiz
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	Fork() Exit() Wait()

Process	Windows	Quiz
File manipulation	CreateFile() ReadFile() WriteFile()	Open() Read() Write() Close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() Read() Write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	Getpid() Alarm() Sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() Shmget() Mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorgroup()	Chmod() Umask() Chown()

2. CPU Scheduling

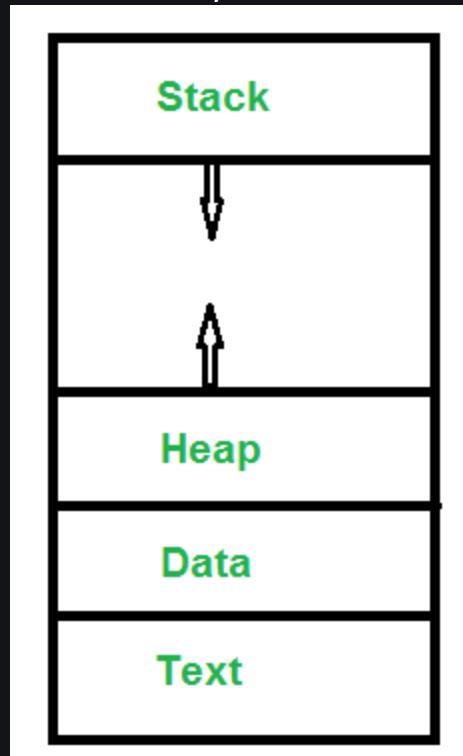
A process is a program in execution. For example, when we write a program in C or C++ and compile it, the [compiler](#) creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

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).

Some of the systems call in this category are as follows.

1. Create a child’s process identical to the parent’s.
2. Terminate a process
3. Wait for a child process to terminate
4. Change the priority of the 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

What does a process look like in memory?



Explanation of Process

1. **Text Section:** A Process, sometimes known as the Text Section, also includes the current activity represented by the value of the Program Counter.
2. **Stack:** The stack contains temporary data, such as function parameters, returns addresses, and local variables.
3. **Data Section:** Contains the global variable.
4. **Heap Section:** Dynamically allocated memory to process during its run time.

Refer to [this](#) for more details on sections.

Attributes or Characteristics of a Process

A process has the following attributes.

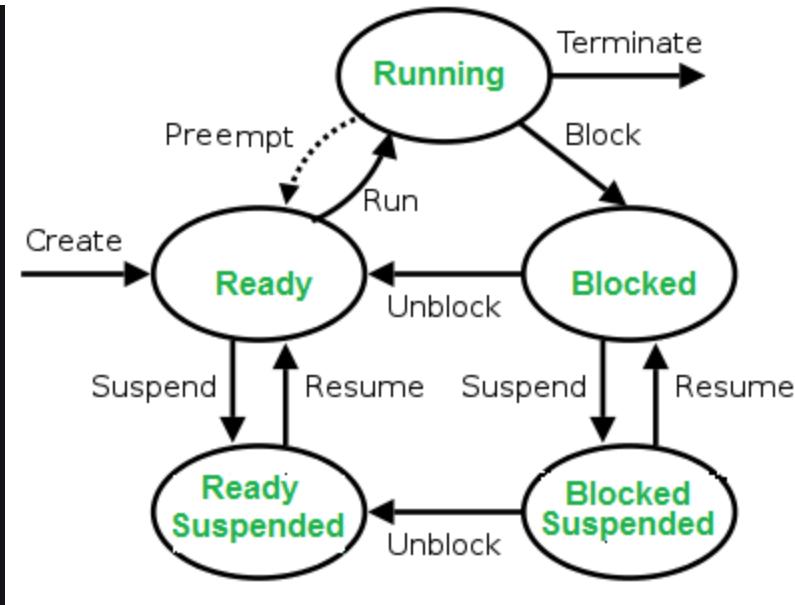
1. **Process Id:** A unique identifier assigned by the operating system
2. **Process State:** Can be ready, running, etc.
3. **CPU registers:** Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of the CPU)
4. **Accounts information:** Amount of CPU 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 PCB. All of the above attributes are part of the PCB.

States of Process

A process is in one of the following states:

1. **New:** Newly Created Process (or) being-created process.
2. **Ready:** After the creation process moves to the 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 a suspended ready state
7. **Suspended Block:** When the waiting queue becomes full.



Context Switching: The process of saving the context of one process and loading the context of another process is known as Context Switching. In simple terms, it is like loading and unloading the process from the running state to the ready state.

When Does Context Switching Happen?

1. When a high-priority process comes to a ready state (i.e. with higher priority than the process)
2. An interrupt occurs
3. User and kernel-mode switch (It is not necessary though)
4. Preemptive CPU scheduling is used.

Context Switch vs Mode Switch

A mode switch occurs when the [CPU](#) privilege level is changed, for example when a system call is made or a fault occurs. The kernel works in more a privileged mode than a standard user task. If a user process wants to access things that are only accessible to the kernel, a mode switch must occur. The currently executing process need not be changed during a mode switch. A mode switch typically occurs for a process context switch to occur. Only the [kernel](#) can cause a context switch.

CPU-Bound vs I/O-Bound Processes

A CPU-bound process requires more CPU time or spends more time in the running state. An I/O-bound process requires more I/O time and less CPU time. An I/O-bound process spends more time in the waiting state.

Process planning is an integral part of the process management operating system. It refers to the mechanism used by the operating system to determine which process to run next. The goal of process scheduling is to improve overall system performance by maximizing CPU utilization, minimizing execution time, and improving system response time.

Scheduling Algorithms

The operating system can use different scheduling algorithms to schedule processes. Here are some commonly used timing algorithms:

1. **First-come, first-served (FCFS):** This is the simplest scheduling algorithm, where the process is executed on a first-come, first-served basis. FCFS is non-preemptive, which means that once a process starts executing, it continues until it is finished or waiting for I/O.
2. **Shortest Job First (SJF):** SJF is a proactive scheduling algorithm that selects the process with the shortest burst time. The burst time is the time a process takes to complete its execution. SJF minimizes the average waiting time of processes.
3. **Round Robin (RR):** RR is a proactive scheduling algorithm that reserves a fixed amount of time in a round for each process. If a process does not complete its execution within the specified time, it is blocked and added to the end of the queue. RR ensures fair distribution of CPU time to all processes and avoids starvation.
4. **Priority Scheduling:** This scheduling algorithm assigns priority to each process and the process with the highest priority is executed first. Priority can be set based on process type, importance, or resource requirements.
5. **Multilevel queue:** This scheduling algorithm divides the ready queue into several separate queues, each queue having a different priority. Processes are queued based on their priority, and each queue uses its own scheduling algorithm. This scheduling algorithm is useful in scenarios where different types of processes have different priorities.

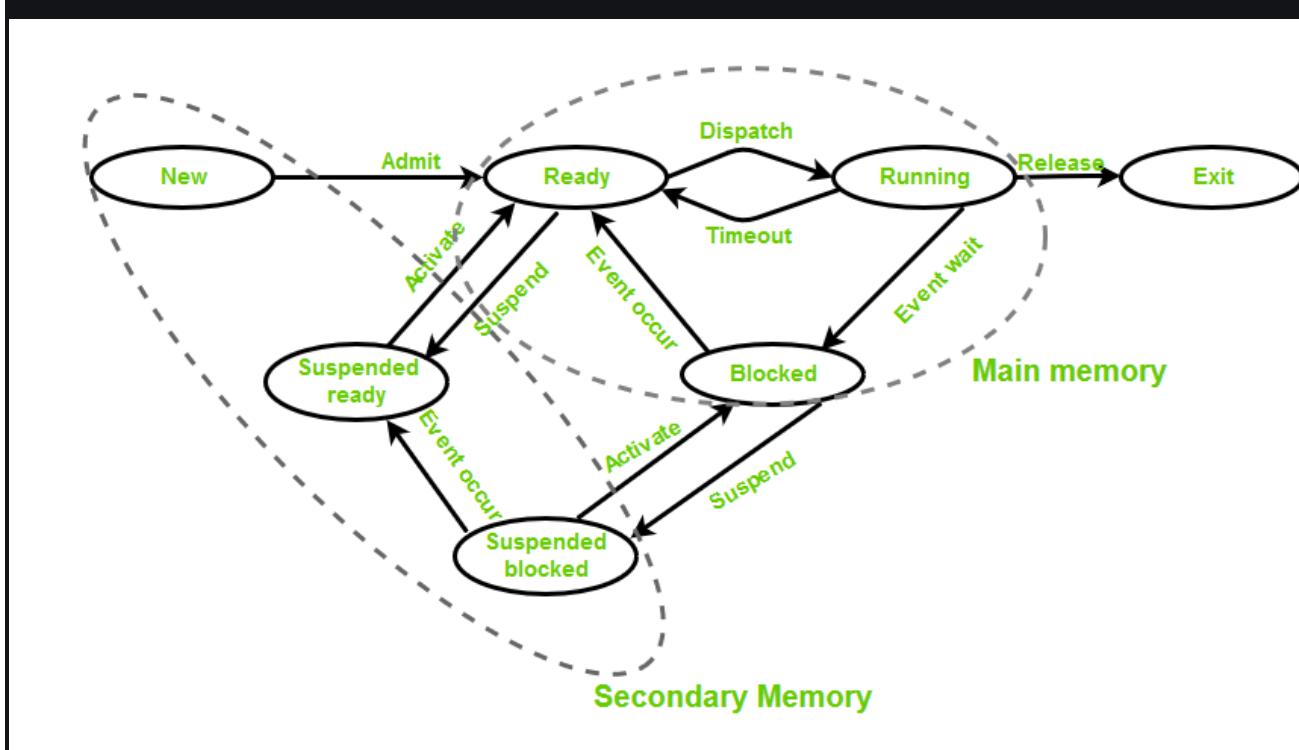
A process has several stages that it passes through from beginning to end. There must be a minimum of five states. Even though during execution, the process could be in one of these states, the names of the states are not standardized. Each process goes through several stages throughout its life cycle.

Process States in Operating System

The states of a process are as follows:

- **New (Create):** In this step, the process is about to be created but not yet created. It is the program that 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 called ready queue for ready processes.
- **Run:** The process is chosen from the ready queue by the 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 waits for the 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. The resources allocated to the process will be released or deallocated.
- **Suspend Ready:** Process that was initially in the ready state but was swapped out of main memory(refer to Virtual Memory topic) and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a 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.

How does a process move between different states in an operating system?

A process can move between different states in an operating system based on its

execution status and resource availability. Here are some examples of how a process can move between different states:

- **New to ready:** When a process is created, it is in a new state. It moves to the ready state when the operating system has allocated resources to it and it is ready to be executed.
- **Ready to running:** When the CPU becomes available, the operating system selects a process from the ready queue depending on various scheduling algorithms and moves it to the running state.
- **Running to blocked:** When a process needs to wait for an event to occur (I/O operation or system call), it moves to the blocked state. For example, if a process needs to wait for user input, it moves to the blocked state until the user provides the input.
- **Running to ready:** When a running process is preempted by the operating system, it moves to the ready state. For example, if a higher-priority process becomes ready, the operating system may preempt the running process and move it to the ready state.
- **Blocked to ready:** When the event a blocked process was waiting for occurs, the process moves to the ready state. For example, if a process was waiting for user input and the input is provided, it moves to the ready state.
- **Running to terminated:** When a process completes its execution or is terminated by the operating system, it moves to the terminated state.

2.1 Types of Schedulers

1. **Long-term – performance:** Decides 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 which also indicates that it runs infrequently. Hence it is called a long-term scheduler.
2. **Short-term – Context switching time:** Short-term scheduler will decide which process is to be executed next and then it will call the dispatcher. A dispatcher is a software that moves the process from ready to run and vice versa. In other words, it is context switching. It runs frequently. Short-term scheduler is also called CPU scheduler.
3. **Medium-term – Swapping time:** Suspension decision is taken by the medium-term scheduler. The medium-term scheduler is used for swapping which is moving the process from main memory to secondary and vice versa. The swapping is done to reduce degree of multiprogramming.

Multiprogramming

We have many processes ready to run. There are two types of multiprogramming:

1. **Preemption** – Process is forcefully removed from CPU. Pre-emotion is also called time sharing or multitasking.

2. **Non-preemption** – Processes are not removed until they complete the execution. Once control is given to the CPU for a process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

Degree of Multiprogramming

The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100, this means 100 processes can reside in the ready state at maximum.

Operation on the Process

1. **Creation:** The process will be ready once it has been created, enter the ready queue (main memory), and be prepared for execution.

2. **Planning:** The operating system picks one process to begin executing from among the numerous processes that are currently in the ready queue. Scheduling is the process of choosing the next process to run.

3. **Application:** The processor begins running the process as soon as it is scheduled to run. During execution, a process may become blocked or wait, at which point the processor switches to executing the other processes.

4. **Killing or Deletion:** The OS will terminate the process once its purpose has been fulfilled. The process's context will be over there.

5. **Blocking:** When a process is waiting for an event or resource, it is blocked. The operating system will place it in a blocked state, and it will not be able to execute until the event or resource becomes available.

6. **Resumption:** When the event or resource that caused a process to block becomes available, the process is removed from the blocked state and added back to the ready queue.

7. **Context Switching:** When the operating system switches from executing one process to another, it must save the current process's context and load the context of the next process to execute. This is known as context switching.

8. **Inter-Process Communication:** Processes may need to communicate with each other to share data or coordinate actions. The operating system provides mechanisms for inter-process communication, such as shared memory, message passing, and synchronization primitives.

9. **Process Synchronization:** Multiple processes may need to access a shared resource or critical section of code simultaneously. The operating system provides synchronization mechanisms to ensure that only one process can access the resource or critical section at a time.

10. **Process States:** Processes may be in one of several states, including ready, running, waiting, and terminated. The operating system manages the process states and transitions between them.

2.2 Features of the Process State

- A process can move from the running state to the waiting state if it needs to wait for a resource to become available.

- A process can move from the waiting state to the ready state when the resource it was waiting for becomes available.
- A process can move from the ready state to the running state when it is selected by the operating system for execution.
- The scheduling algorithm used by the operating system determines which process is selected to execute from the ready state.
- The operating system may also move a process from the running state to the ready state to allow other processes to execute.
- A process can move from the running state to the terminated state when it completes its execution.
- A process can move from the waiting state directly to the terminated state if it is aborted or killed by the operating system or another process.
- A process can go through ready, running and waiting state any number of times in its lifecycle but new and terminated happens only once.
- The process state includes information about the program counter, CPU registers, memory allocation, and other resources used by the process.
- The operating system maintains a process control block (PCB) for each process, which contains information about the process state, priority, scheduling information, and other process-related data.
- The process state diagram is used to represent the transitions between different states of a process and is an essential concept in process management in operating systems.

While creating a process, the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the [process control block](#) (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, [scheduling algorithms](#), etc.

All this information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB. A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, that means logically contains a [PCB](#) for all of the current processes in the system.

Pointer

Process State

Process Number

Program Counter

Registers

Memory Limits

Open File Lists

**Misc. Accounting
and Status Data**

Process Control Block

1. **Pointer:** It is a stack pointer that is required to be saved when the process is switched from one state to another to retain the current position of the process.
2. **Process state:** It stores the respective state of the process.
3. **Process number:** Every process is assigned a unique id known as process ID or PID which stores the process identifier.
4. **Program counter:** It stores the counter, which contains the address of the next instruction that is to be executed for the process.
5. **Register:** These are the CPU registers which include the accumulator, base, registers, and general-purpose registers.
6. **Memory limits:** This field contains the information about memory management system used by the operating system. This may include page tables, segment tables, etc.
7. **Open files list :** This information includes the list of files opened for a process.

2.3

Additional Points to Consider for Process Control Block (PCB)

- **Interrupt handling:** The PCB also contains information about the interrupts that a process may have generated and how they were handled by the operating system.
- **Context switching:** The process of switching from one process to another is called context switching. The PCB plays a crucial role in context switching by saving the state of the current process and restoring the state of the next process.
- **Real-time systems:** Real-time operating systems may require additional information in the PCB, such as deadlines and priorities, to ensure that time-critical processes are executed in a timely manner.
- **Virtual memory management:** The PCB may contain information about a process's virtual memory management, such as page tables and page fault handling.
- **Inter-process communication:** The PCB can be used to facilitate inter-process communication by storing information about shared resources and communication channels between processes.
- **Fault tolerance:** Some operating systems may use multiple copies of the PCB to provide fault tolerance in case of hardware failures or software errors.

2.4

Advantages-

1. **Efficient process management:** The process table and PCB provide an efficient way to manage processes in an operating system. The process table contains all the information about each process, while the PCB contains the current state of the process, such as the program counter and CPU registers.
2. **Resource management:** The process table and PCB allow the operating system to manage system resources, such as memory and CPU time, efficiently. By keeping track of each process's resource usage, the operating system can ensure that all processes have access to the resources they need.

3. **Process synchronization:** The process table and PCB can be used to synchronize processes in an operating system. The PCB contains information about each process's synchronization state, such as its waiting status and the resources it is waiting for.
4. **Process scheduling:** The process table and PCB can be used to schedule processes for execution. By keeping track of each process's state and resource usage, the operating system can determine which processes should be executed next.

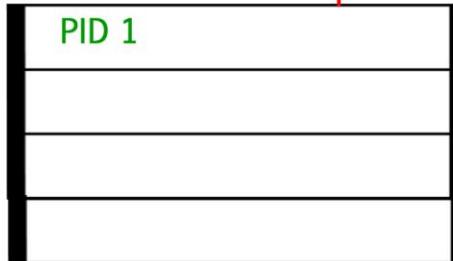
2.5 Disadvantages-

1. **Overhead:** The process table and PCB can introduce overhead and reduce system performance. The operating system must maintain the process table and PCB for each process, which can consume system resources.
2. **Complexity:** The process table and PCB can increase system complexity and make it more challenging to develop and maintain operating systems. The need to manage and synchronize multiple processes can make it more difficult to design and implement system features and ensure system stability.
3. **Scalability:** The process table and PCB may not scale well for large-scale systems with many processes. As the number of processes increases, the process table and PCB can become larger and more difficult to manage efficiently.
4. **Security:** The process table and PCB can introduce security risks if they are not implemented correctly. Malicious programs can potentially access or modify the process table and PCB to gain unauthorized access to system resources or cause system instability.
5. **Miscellaneous accounting and status data** – This field includes information about the amount of CPU used, time constraints, jobs or process number, etc. The process control block stores the register content also known as execution content of the processor when it was blocked from running. This execution content architecture enables the operating system to restore a process's execution context when the process returns to the running state. When the process makes a transition from one state to another, the operating system updates its information in the process's PCB. The operating system maintains pointers to each process's PCB in a process table so that it can access the PCB

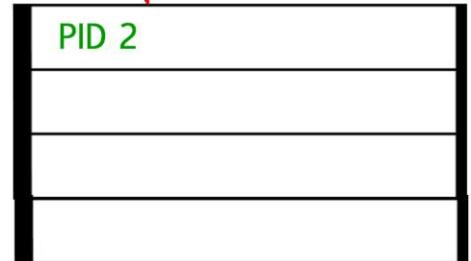
quickly.

PID	PCB
1	
2	
3	

PCB



PCB



Process table and process control block

2.6 Categories in Scheduling

Scheduling falls into one of two categories:

- **Non-preemptive:** In this case, a process's resource cannot be taken before the process has finished running. When a running process finishes and transitions to a waiting state, resources are switched.
- **Preemptive:** In this case, the OS assigns resources to a process for a predetermined period of time. The process switches from running state to ready state or from waiting for state to ready state during resource allocation. This switching happens because the CPU may give other processes priority and substitute the currently active process for the higher priority process.

There are three types of process schedulers.

2.7 Long Term or Job Scheduler

It brings the new process to the 'Ready State'. It controls the **Degree of Multiprogramming**, 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.

2.8 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 no starvation due 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.

2.9 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](#).

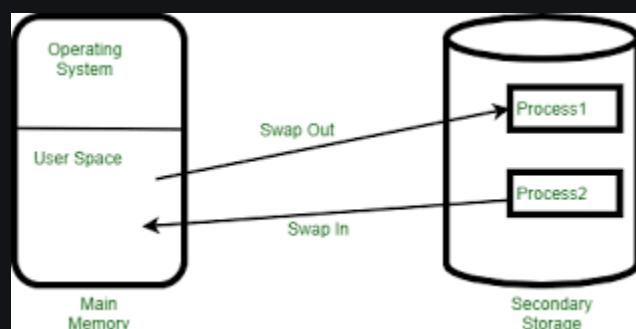
2.10 Some Other Schedulers

- **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).

2.11 Context Switching

In order for a process execution to be continued from the same point at a later time, context switching is a mechanism to store and restore the state or context of a CPU in the Process Control block. A context switcher makes it possible for multiple processes to share a single CPU using this method. A multitasking operating system must include context switching among its features.

The state of the currently running process is saved into the process control block when the scheduler switches the CPU from executing one process to another. The state used to set the PC, registers, etc. for the process that will run next is then loaded from its own PCB. After that, the second can start processing.



In order for a process execution to be continued from the same point at a later time, context switching is a mechanism to store and restore the state or context of a CPU in the [Process Control block](#). A context switcher makes it possible for multiple processes to share a single CPU using this method. A multitasking operating system must include context switching among its features.

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

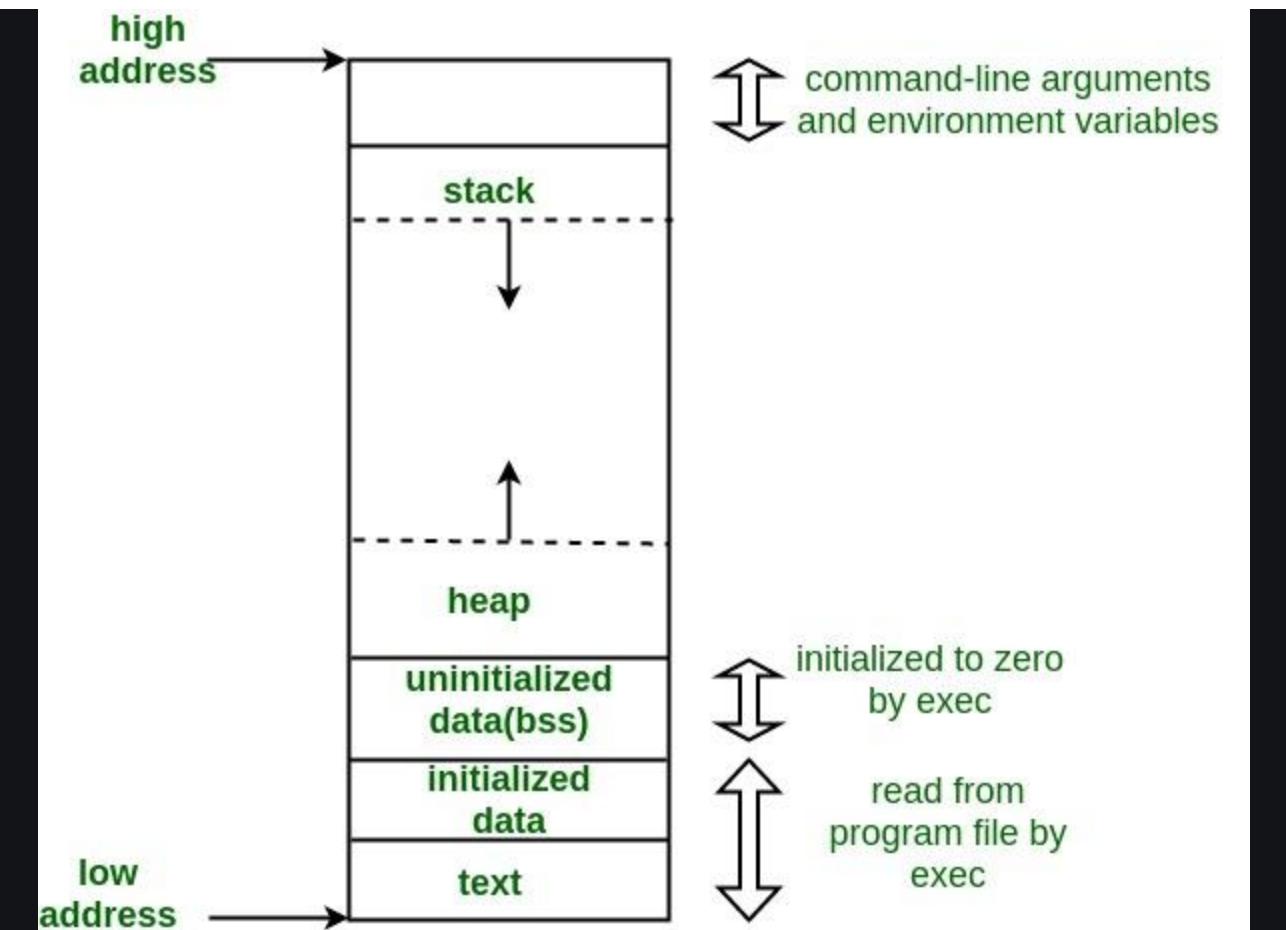
2.12 What is a process?

In computing, a process is **the instance of a computer program that is being executed by one or many threads**. It contains the program code and its activity. Depending on the [operating system](#) (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

2.13 How is process memory used for efficient operation?

The [process memory](#) is divided into four sections for efficient operation:

- The **text category** is composed of integrated program code, which is read from fixed storage when the program is launched.
- The **data class** is made up of global and static variables, distributed and executed before the main action.
- [Heap](#) is used for flexible, or dynamic memory allocation and is managed by calls to new, delete, malloc, free, etc.
- The [stack](#) is used for local variables. The space in the stack is reserved for local variables when it is announced.



To know further, you can refer to our detailed article on [States of a Process in Operating system](#).

2.14 What is Process Scheduling?

Process Scheduling is the process of the process manager handling the removal of an active process from the CPU and selecting another process based on a specific strategy.

Process Scheduling is an integral part of Multi-programming applications. Such operating systems allow more than one process to be loaded into usable memory at a time and the loaded shared CPU process uses repetition time.

There are three types of [process schedulers](#):

- Long term or Job Scheduler
- Short term or CPU Scheduler
- Medium-term Scheduler

2.15 Why do we need to schedule processes?

- **Scheduling** is important in many different computer environments. One of the most important areas is scheduling which programs will work on the CPU. This task is handled by the Operating System (OS) of the computer and there are many different ways in which we can choose to configure programs.
- **Process Scheduling** allows the OS to allocate CPU time for each process. Another important reason to use a process scheduling system is that it keeps

- the CPU busy at all times. This allows you to get less response time for programs.
- Considering that there may be hundreds of programs that need to work, the OS must launch the program, stop it, switch to another program, etc. The way the OS configures the system to run another in the CPU is called “[context switching](#)”. If the OS keeps context-switching programs in and out of the provided CPUs, it can give the user a tricky idea that he or she can run any programs he or she wants to run, all at once.
 - So now that we know we can run 1 program at a given CPU, and we know we can change the operating system and remove another one using the context switch, how do we choose which programs we need. run, and with what program?
 - That’s where **scheduling** comes in! First, you determine the metrics, saying something like “the amount of time until the end”. We will define this metric as “the time interval between which a function enters the system until it is completed”. Second, you decide on a metrics that reduces metrics. We want our tasks to end as soon as possible.

2.16 What is the need for CPU scheduling algorithm?

CPU scheduling is the process of deciding which process will own the CPU to use while another process is suspended. The main function of the CPU scheduling is to ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.

In [Multiprogramming](#), if the long-term scheduler selects multiple I / O binding processes then most of the time, the CPU remains an idle. The function of an effective program is to improve resource utilization.

If most operating systems change their status from performance to waiting then there may always be a chance of failure in the system. So in order to minimize this excess, the OS needs to schedule tasks in order to make full use of the CPU and avoid the possibility of deadlock.

Objectives of Process Scheduling Algorithm:

- Utilization of CPU at maximum level. **Keep CPU as busy as possible.**
- Allocation of CPU should be fair.**
- Throughput should be Maximum.** i.e. Number of processes that complete their execution per time unit should be maximized.
- Minimum turnaround time**, i.e. time taken by a process to finish execution should be the least.
- There should be a **minimum waiting time** and the process should not starve in the ready queue.
- Minimum response time.** It means that the time when a process produces the first response should be as less as possible.

2.17 What are the different terminologies to take care of in any CPU Scheduling algorithm?

- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

2.18 Things to take care while designing a CPU Scheduling algorithm?

Different **CPU Scheduling algorithms** have different structures and the choice of a particular algorithm depends on a variety of factors. Many conditions have been raised to compare CPU scheduling algorithms.

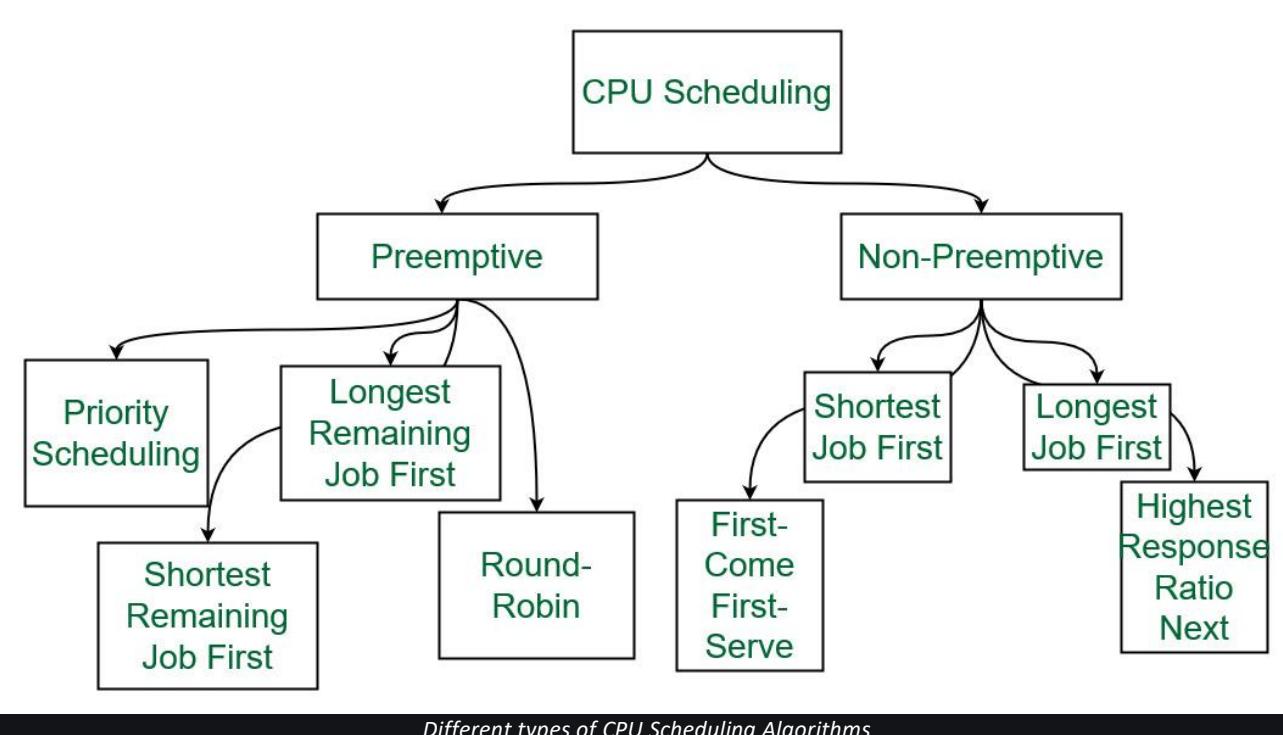
The criteria include the following:

- **CPU utilization:** The main purpose of any CPU algorithm is to keep the CPU as busy as possible. Theoretically, CPU usage can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the system load.
- **Throughput:** The average CPU performance is the number of processes performed and completed during each unit. This is called throughput. The output may vary depending on the length or duration of the processes.
- **Turn round Time:** For a particular process, the important conditions are how long it takes to perform that process. The time elapsed from the time of process delivery to the time of completion is known as the conversion time. Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I / O.
- **Waiting Time:** The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.
- **Response Time:** In a collaborative system, turn around time is not the best option. The process may produce something early and continue to computing the new results while the previous results are released to the user. Therefore another method is the time taken in the submission of the application process until the first response is issued. This measure is called response time.

2.19 What are the different types of CPU Scheduling Algorithms?

There are mainly two types of scheduling methods:

- [Preemptive Scheduling](#): Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- [Non-Preemptive Scheduling](#): Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.



Different types of CPU Scheduling Algorithms

Let us now learn about these CPU scheduling algorithms in operating systems one by one:

1. First Come First Serve:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using [FIFO queue](#).

Characteristics of FCFS:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS:

- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

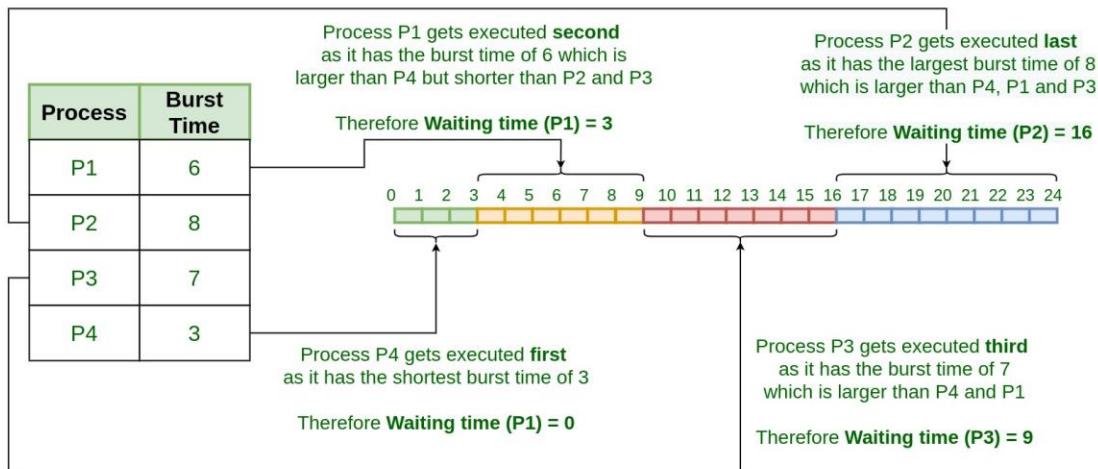
- FCFS suffers from **Convoy effect**.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [First come, First serve Scheduling](#).

2. Shortest Job First(SJF):

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Shortest Job First (SJF) Scheduling Algorithm



Characteristics of SJF:

- Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages of Shortest Job first:

- As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm.
- SJF is generally used for long term scheduling

Disadvantages of SJF:

- One of the demerit SJF has is starvation.
- Many times it becomes complicated to predict the length of the upcoming CPU request

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [Shortest Job First](#).

3. Longest Job First(LJF):

Longest Job First(LJF) scheduling process is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first. Longest Job First is non-preemptive in nature.

Characteristics of LJF:

- Among all the processes waiting in a waiting queue, CPU is always assigned to the process having largest burst time.
- If two processes have the same burst time then the tie is broken using [FCFS](#) i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of LJF:

- No other task can schedule until the longest job or process executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of LJF:

- Generally, the LJF algorithm gives a very high [average waiting time](#) and [average turn-around time](#) for a given set of processes.
- This may lead to convoy effect.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on the [Longest job first scheduling](#).

4. Priority Scheduling:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of [CPU scheduling algorithm](#) that works **based on the priority** of a process. In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there are more than one processor with equal value, then the most important CPU planning algorithm works on the basis of the FCFS (First Come First Serve) algorithm.

Characteristics of Priority Scheduling:

- Schedules tasks based on priority.
- When the higher priority work arrives while a task with less priority is executed, the higher priority work takes the place of the less priority one and
- The latter is suspended until the execution is complete.
- Lower is the number assigned, higher is the priority level of a process.

Advantages of Priority Scheduling:

- The average waiting time is less than FCFS
- Less complex

Disadvantages of Priority Scheduling:

- One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the [Starvation Problem](#). This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [Priority Preemptive Scheduling algorithm](#).

5. Round robin:

Round Robin is a [CPU scheduling algorithm](#) where each process is cyclically assigned a fixed time slot. It is the [preemptive](#) version of [First come First Serve CPU Scheduling algorithm](#). Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Characteristics of Round robin:

- It's simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as the processes are given to the CPU for a very limited time.

Advantages of Round robin:

- Round robin seems to be fair as every process gets an equal share of CPU.
- The newly created process is added to the end of the ready queue.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on the [Round robin Scheduling algorithm](#).

6. Shortest Remaining Time First:

Shortest remaining time first is the preemptive version of the Shortest job first which we have discussed earlier where the processor is allocated to the job closest to completion. In SRTF the process with the smallest amount of time remaining until completion is selected to execute.

Characteristics of Shortest remaining time first:

- SRTF algorithm makes the processing of the jobs faster than SJF algorithm, given it's overhead charges are not counted.
- The context switch is done a lot more times in SRTF than in SJF and consumes the CPU's valuable time for processing. This adds up to its processing time and diminishes its advantage of fast processing.

Advantages of SRTF:

- In SRTF the short processes are handled very fast.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.

Disadvantages of SRTF:

- Like the shortest job first, it also has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on the [shortest remaining time first](#).

7. Longest Remaining Time First:

The longest remaining time first is a preemptive version of the longest job first scheduling algorithm. This scheduling algorithm is used by the operating system to

program incoming processes for use in a systematic way. This algorithm schedules those processes first which have the longest processing time remaining for completion.

Characteristics of longest remaining time first:

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.
- If two processes have the same burst time then the tie is broken using [FCFS](#) i.e. the process that arrived first is processed first.
- LRTF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of LRTF:

- No other process can execute until the longest task executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of LRTF:

- This algorithm gives a very high [average waiting time](#) and [average turn-around time](#) for a given set of processes.
- This may lead to a convoy effect.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on the [longest remaining time first](#).

8. Highest Response Ratio Next:

Highest Response Ratio Next is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithms. The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Characteristics of Highest Response Ratio Next:

- The **criteria** for HRRN is **Response Ratio**, and the **mode** is **Non-Preemptive**.
- HRRN is considered as the modification of [Shortest Job First](#) to reduce the problem of [starvation](#).
- In comparison with SJF, during the HRRN scheduling algorithm, the CPU is allotted to the next process which has the **highest response ratio** and not to the process having less burst time.

Advantages of HRRN:

- HRRN Scheduling algorithm generally gives better performance than the [shortest job first](#) Scheduling.
- There is a reduction in waiting time for longer jobs and also it encourages shorter jobs.

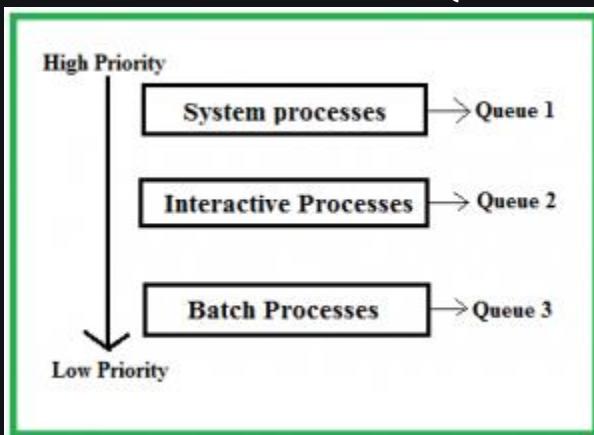
Disadvantages of HRRN:

- The implementation of HRRN scheduling is not possible as it is not possible to know the burst time of every job in advance.
- In this scheduling, there may occur an overload on the CPU.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [Highest Response Ratio Next](#).

9. Multiple Queue Scheduling:

Processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process and a **background (batch)** process. These two classes have different scheduling needs. For this kind of situation **Multilevel Queue Scheduling** is used.



The description of the processes in the above diagram is as follows:

- **System Processes:** The CPU itself has its process to run, generally termed as System Process.
- **Interactive Processes:** An Interactive Process is a type of process in which there should be the same type of interaction.
- **Batch Processes:** Batch processing is generally a technique in the Operating system that collects the programs and data together in the form of a **batch** before the **processing** starts.

Advantages of multilevel queue scheduling:

- The main merit of the multilevel queue is that it has a low scheduling overhead.

Disadvantages of multilevel queue scheduling:

- Starvation problem
- It is inflexible in nature

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [Multilevel Queue Scheduling](#).

10. Multilevel Feedback Queue Scheduling::

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like **Multilevel Queue Scheduling** but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics of Multilevel Feedback Queue Scheduling:

- In a [multilevel queue-scheduling](#) algorithm, processes are permanently assigned to a queue on entry to the system, and processes are not allowed to move between queues.
- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,

- But on the other hand disadvantage of being inflexible.

Advantages of Multilevel feedback queue scheduling:

- It is more flexible
- It allows different processes to move between different queues

Disadvantages of Multilevel feedback queue scheduling:

- It also produces CPU overheads
- It is the most complex algorithm.

To learn about how to implement this CPU scheduling algorithm, please refer to our detailed article on [Multilevel Feedback Queue Scheduling](#).

2.20 Comparison between various CPU Scheduling algorithms

Here is a brief comparison between different CPU scheduling algorithms:

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
FCFS	According to the arrival time of the processes, the CPU is allocated.	Simple and easy to implement	Large.	No	No	Slow performance
SJF	Based on the lowest CPU burst time (BT).	More complex than FCFS	Smaller than FCFS	No	Yes	Minimum Average Waiting Time
LJFS	Based on the highest	More complex	Depending on some	No	Yes	Big turn-around time

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	CPU burst time (BT)	than FCFS	measures e.g., arrival time, process size, etc.			
LRTF	Same as LJFS the allocation of the CPU is based on the highest CPU burst time (BT). But it is preemptive	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc.	Yes	Yes	The preference is given to the longer jobs
SRTF	Same as SJF the allocation of the CPU is based on the lowest CPU burst	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc	Yes	Yes	The preference is given to the short jobs

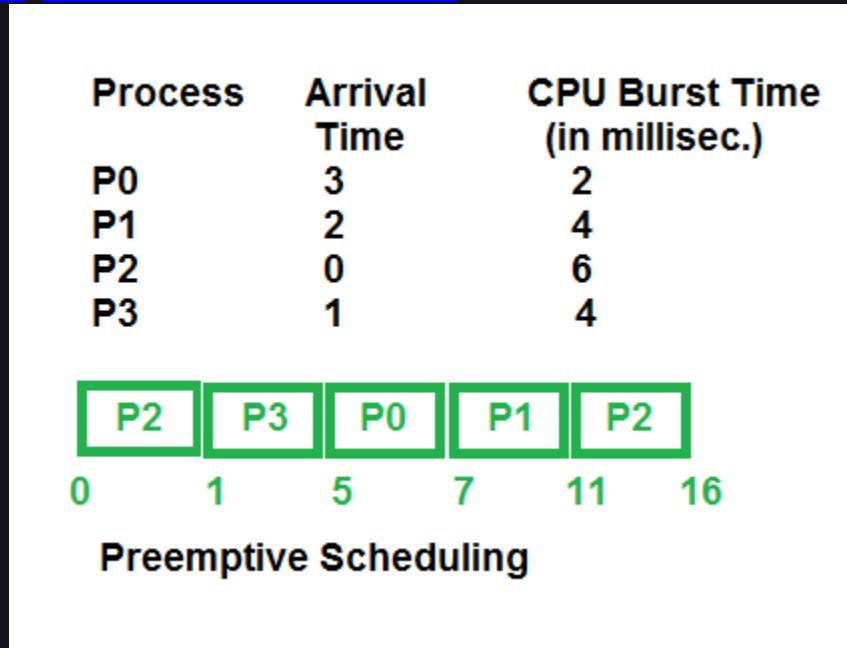
Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	time (BT). But it is preemptive.					
RR	According to the order of the process arrives with fixed time quantum (TQ)	The complexity depends on Time Quantum size	Large as compared to SJF and Priority scheduling.	Yes	No	Each process has given a fairly fixed time
Priority Pre-emptive	According to the priority. The bigger priority task executes first	This type is less complex	Smaller than FCFS	Yes	Yes	Well performance but contain a starvation problem
Priority non-preemptive	According to the priority with	This type is less complex than Priority	Preemptive Smaller than FCFS	No	Yes	Most beneficial with batch systems

Algorithm	Allocation	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	monitoring the new incoming higher priority jobs	preemptive				
MLQ	According to the process that resides in the bigger queue priority	More complex than the priority scheduling algorithms	Smaller than FCFS	No	Yes	Good performance but contain a starvation problem
MFLQ	According to the process of a bigger priority queue.	It is the most Complex but its complexity rate depends on the TQ size	Smaller than all scheduling types in many cases	No	No	Good performance

2.21 Preemptive Scheduling

Preemptive scheduling is used when a process switches from the running state to the ready state or from the waiting state to the ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Algorithms based on preemptive scheduling are [Round Robin \(RR\)](#), [Shortest Remaining Time First \(SRTF\)](#), [Priority \(preemptive version\)](#), etc.



Preemptive scheduling has a number of advantages and disadvantages. The following are non-preemptive scheduling's benefits and drawbacks:

Advantages

1. Because a process may not monopolize the processor, it is a more reliable method.
2. Each occurrence prevents the completion of ongoing tasks.
3. The average response time is improved.
4. Utilizing this method in a multi-programming environment is more advantageous.
5. The operating system makes sure that every process using the CPU is using the same amount of CPU time.

Disadvantages

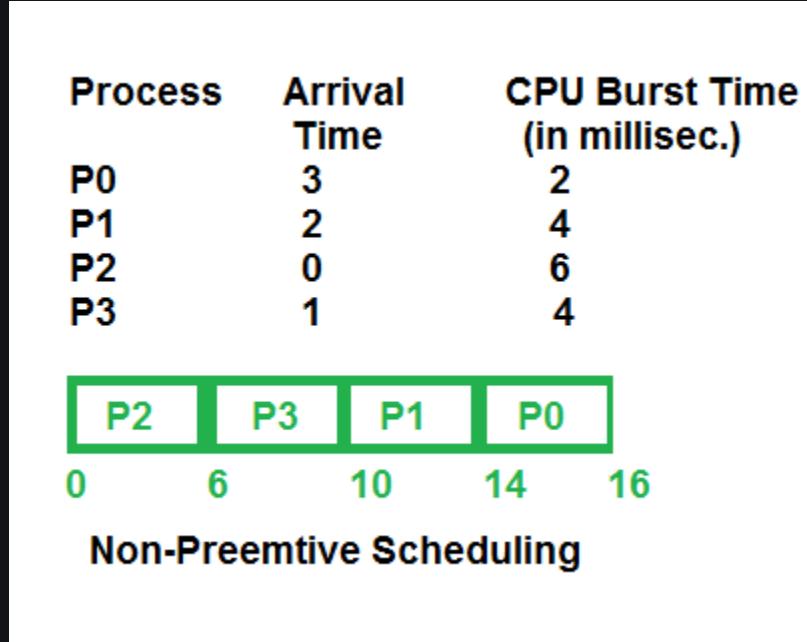
1. Limited computational resources must be used.
2. Suspending the running process, change the context, and dispatch the new incoming process all take more time.
3. The low-priority process would have to wait if multiple high-priority processes arrived at the same time.

2.22 Non-Preemptive Scheduling

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU

in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

Algorithms based on non-preemptive scheduling are: [Shortest Job First \(SJF basically non preemptive\)](#) and [Priority \(nonpreemptive version\)](#), etc.



Non-preemptive scheduling has both advantages and disadvantages. The following are non-preemptive scheduling's benefits and drawbacks:

Advantages

1. It has a minimal scheduling burden.
2. It is a very easy procedure.
3. Less computational resources are used.
4. It has a high throughput rate.

Disadvantages

1. Its response time to the process is super.
2. Bugs can cause a computer to freeze up.

2.23 Key Differences Between Preemptive and Non-Preemptive Scheduling

1. In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when a higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.

3. In Preemptive Scheduling, there is the overhead of switching the process from the ready state to the running state, vice-versa, and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. , in non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with a small burst time may have to starve.
5. Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive in the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative which is not the case with Non-preemptive Scheduling.

Comparison Chart

Parameter	PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
Basic	In this resources(CPU Cycle) are allocated to a process for a limited time.	Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted until it terminates itself or its time is up.
Starvation	If a process having high priority frequently arrives in the ready queue, a low priority process may starve.	If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve.
Overhead	It has overheads of scheduling the processes.	It does not have overheads.

Parameter	PREEEMPTIVE SCHEDULING	NON-PREEEMPTIVE SCHEDULING
Flexibility	flexible	rigid
Cost	cost associated	no cost associated
CPU Utilization	In preemptive scheduling, CPU utilization is high.	It is low in non preemptive scheduling.
Waiting Time	Preemptive scheduling waiting time is less.	Non-preemptive scheduling waiting time is high.
Response Time	Preemptive scheduling response time is less.	Non-preemptive scheduling response time is high.
Decision making	Decisions are made by the scheduler and are based on priority and time slice allocation	Decisions are made by the process itself and the OS just follows the process's instructions
Process control	The OS has greater control over the scheduling of processes	The OS has less control over the scheduling of processes
Overhead	Higher overhead due to frequent context switching	Lower overhead since context switching is less frequent
Examples	Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First.	Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First.

A *Context switch* is a time spent between two processes (i.e., bringing a waiting process into execution and sending an executing process into a waiting-for state). This happens in

multitasking. The operating system must bring the state information if waiting for the process into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the first and last instructions of the swapping processes. The context switch time is the difference between the two processes.

Let's take an example: Assume there are only two processes, P1 and P2. P1 is executing and P2 is waiting for execution. At some point, the operating system must swap P1 and P2, let's assume it happens at the nth instruction of P1. If $t(x, k)$ indicates the timestamp in microseconds of the kth instruction of process x, then the context switch would take $t(2, 1) - t(1, n)$.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel-level threads that are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t(1, n)$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to avoid these obstacles, we must construct an environment such that after P1 executes, the task scheduler immediately selects P2 to run. This may be accomplished by constructing a data channel, such as a pipe between P1 and P2.

That is, let's allow P1 to be the initial sender and P2 to be the receiver. Initially, P2 is blocked(sleeping) as it awaits the data token. When P1 executes, it delivers the data token over the data channel to P2 and immediately attempts to read the response token. A context switch results and the task scheduler must select another process to run. Since P2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P2 runs, the role of P1 and P2 are swapped. P2 is now acting as the sender and P1 as the blocked receiver.

To summaries –

1. P2 blocks awaiting data from P1
2. P1 marks the starting time.
3. P1 sends a token to P2.
4. P1 attempts to read a response token from P2. This induces a context switch.
5. P2 is scheduled and receives the token.
6. P2 sends a response token to P1.
7. P2 attempts to read a response token from P1. This induces a context switch.
8. P1 is scheduled and receives the token.
9. P1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. In step 2, P1 records the timestamp of the delivery of the

token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T , between these events, may be expressed by:

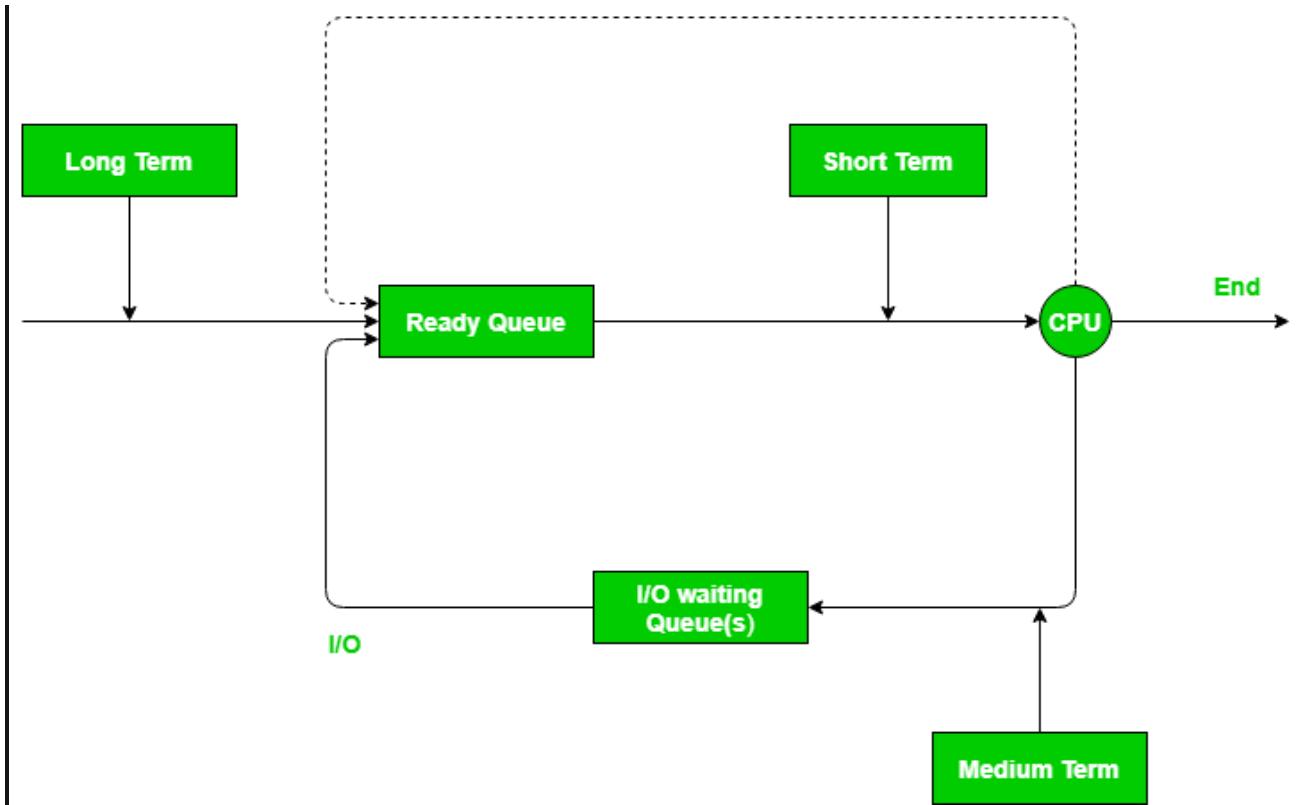
This formula arises because of the following events:

- P1 sends the token (3)
- CPU context switches (4)
- P2 receives it (5)
- P2 then sends the response token (6)
- CPU context switches (7)
- and finally, P1 receives it (8)

Schedulers are special system software that handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

There are three types of Scheduler:

1. **Long-term (job) scheduler** – Due to the smaller size of main memory initially all programs are stored in [secondary memory](#). When they are stored or loaded in the main memory they are called processes. This is the decision of the long-term scheduler to determine how many processes will stay in the ready queue. Hence, in simple words, the long-term scheduler decides the degree of multi-programming of the system.
2. **Medium-term scheduler** – Most often, a running process needs I/O operation which doesn't require a CPU. Hence during the execution of a process when an I/O operation is required then the operating system sends that process from the running queue to the blocked queue. When a process completes its I/O operation then it should again be shifted to the ready queue. ALL these decisions are taken by the medium-term scheduler. Medium-term scheduling is a part of **swapping**.
3. **Short-term (CPU) scheduler** – When there are lots of processes in main memory initially all are present in the ready queue. Among all of the processes, a single process is to be selected for execution. This decision is handled by a short-term scheduler. Let's have a look at the figure given below. It may make a more clear view for you.



Dispatcher – A dispatcher is a special program which comes into play after the scheduler. When the scheduler completes its job of selecting a process, it is the dispatcher which takes that process to the desired state/queue. The dispatcher is the module that gives a process control over the CPU after it has been selected by the short-term scheduler. This function involves the following:

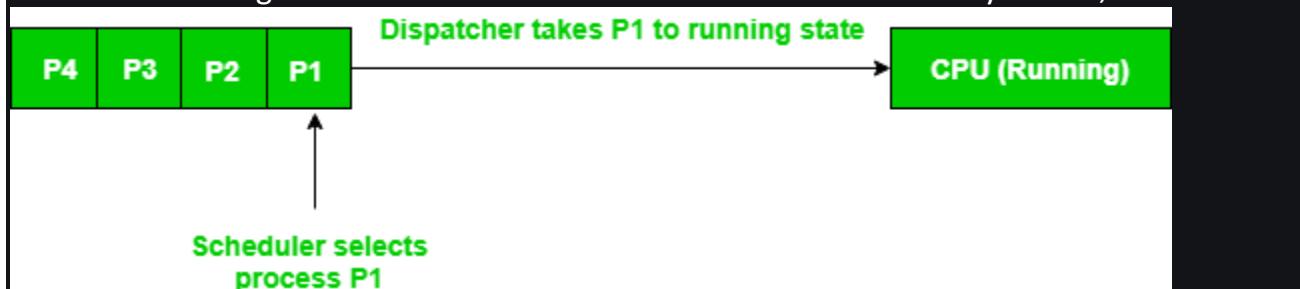
- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

Role of the dispatcher:

1. The dispatcher gives control of the CPU to the process selected by the short-term scheduler.
2. A dispatcher is a communications worker who receives and transmits information to co-ordinate operations of other personnel and vehicles carrying out a service.
3. A dispatcher is a special program which comes into play after the scheduler.
4. When the scheduler completes its job of selecting a process, it is the dispatcher which takes that process to the desired state/ queue.

The Difference between the Scheduler and Dispatcher – Consider a situation, where various processes are residing in the ready queue waiting to be executed. The CPU cannot execute all of these processes simultaneously, so the operating system has to choose a particular process on the basis of the scheduling algorithm used. So, this procedure of selecting a process among various processes is done by the **scheduler**. Once the scheduler

has selected a process from the queue, the **dispatcher** comes into the picture, and it is the dispatcher who takes that process from the ready queue and moves it into the running state. Therefore, the scheduler gives the dispatcher an ordered list of processes which the dispatcher moves to the CPU over time. **Example** – There are 4 processes in the ready queue, P1, P2, P3, P4; Their arrival times are t0, t1, t2, t3 respectively. A First in First out (FIFO) scheduling algorithm is used. Because P1 arrived first, the scheduler will decide it is the first process that should be executed, and the dispatcher will remove P1 from the ready queue and give it to the CPU. The scheduler will then determine P2 to be the next process that should be executed, so when the dispatcher returns to the queue for a new process, it will take P2 and give it to the CPU. This continues in the same way for P3, and then P4.



Properties	DISPATCHER	SCHEDULER
Definition	Dispatcher is a module that gives control of CPU to the process selected by short term scheduler	Scheduler is something which selects a process among various processes
Types	There are no different types in dispatcher. It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term
Dependency	Working of dispatcher is dependent on scheduler. Means dispatcher have to wait until scheduler selects a process.	Scheduler works independently. It works immediately when needed
Algorithm	Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
Time Taken	The time taken by dispatcher is called dispatch latency.	Time taken by scheduler is usually negligible. Hence we neglect it.
Functions	Dispatcher is also responsible for: Context Switching, Switch to	The only work of scheduler is selection of processes.

Properties	DISPATCHER	SCHEDULER
	user mode, Jumping to proper location when process again restarted	
Tasks	Dispatcher allocates the CPU to the process selected by the short-time scheduler.	Scheduler performs three task. Job scheduling (Long-term scheduler), CPU scheduling (Short-term scheduler) and swapping (Medium-term scheduler).
Purpose	To move the process from the ready queue to the CPU	To select the process and decide which process to run
Execution time	It takes a very short execution time	It takes longer execution time than dispatcher
Interaction	The dispatcher works with the CPU and the selected process	The scheduler works with the ready queue and the dispatcher

Convoy Effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.

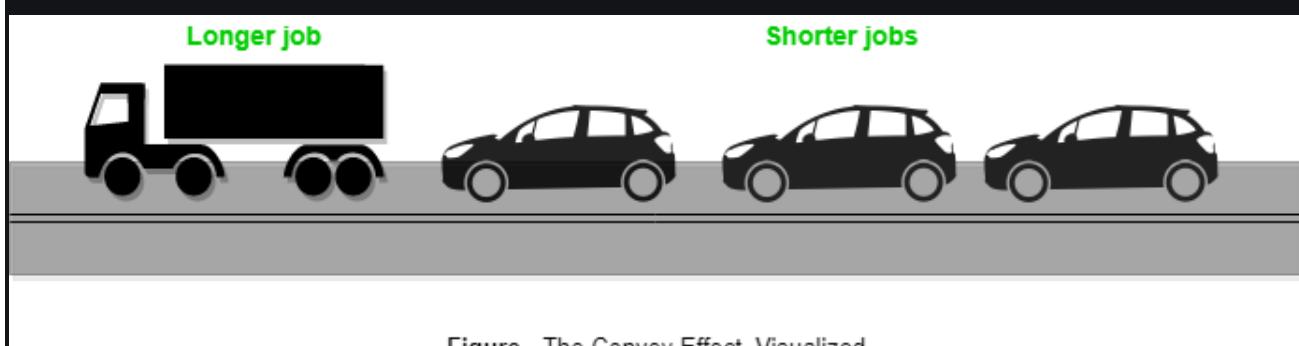


Figure - The Convey Effect, Visualized

FCFS algorithm is non-preemptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation called Convoy Effect.

Suppose there is one CPU intensive (large burst time) process in the ready queue, and several other processes with relatively less burst times but are Input/Output (I/O) bound (Need I/O operations frequently).

Steps are as following below:

- The I/O bound processes are first allocated CPU time. As they are less CPU intensive, they quickly get executed and goto I/O queues.
- Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
- While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
- However, the I/O bound processes are made to wait as the CPU intensive process still hasn't finished. **This leads to I/O devices being idle.**
- When the CPU intensive process gets over, it is sent to the I/O queue so that it can access an I/O device.
- Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
- However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, **the CPU is sitting idle now.**

Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.

To avoid Convoy Effect, preemptive scheduling algorithms like Round Robin Scheduling can be used – as the smaller processes don't have to wait much for CPU time – making their execution faster and leading to less resources sitting idle.

In Operating System, process data is loaded in fixed-sized chunks and each chunk is referred to as a page. The processor loads these pages in fixed-sized chunks of memory called frames. Typically the size of each page is always equal to the frame size.

Page Replacement Algorithms

A page fault occurs when a page is not found in the memory and needs to be loaded from the disk. If a page fault occurs and all memory frames have been already allocated, then the replacement of a page in memory is required on the request of a new page. This is referred to as demand-paging. The choice of which page to replace is specified by page replacement algorithms. The commonly used page replacement algorithms are FIFO, LRU, optimal page replacement algorithms, etc.

Generally, on increasing the number of frames to a process' virtual memory, its execution becomes faster as fewer page faults occur. Sometimes the reverse happens, i.e. more page faults occur when more frames are allocated to a process. This most unexpected result is termed **Belady's Anomaly**.

Bélády's anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

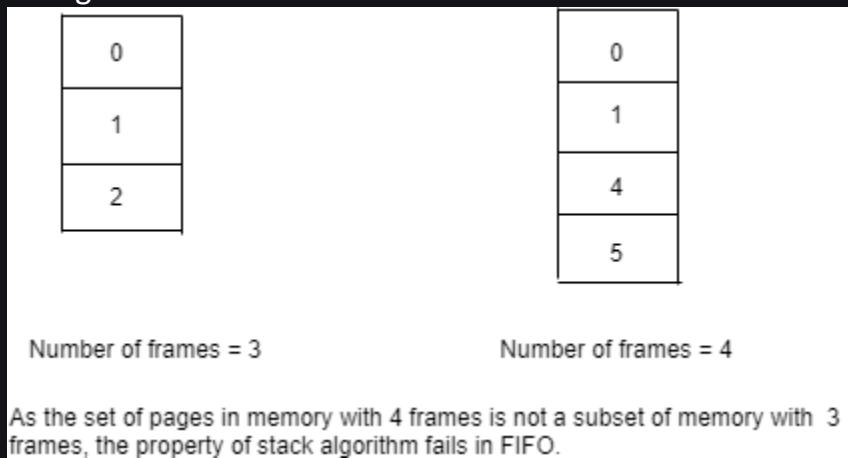
This phenomenon is commonly experienced in the following page replacement algorithms:

1. First in first out (FIFO)
2. Second chance algorithm
3. Random page replacement algorithm

Reason	for	Belady's	Anomaly	-
The other two commonly used page replacement algorithms are Optimal and LRU, but Belady's Anomaly can never occur in these algorithms for any reference string as they belong to a class of stack-based page replacement algorithms.				

A **stack-based algorithm** is one for which it can be shown that the set of pages in memory for N frames is always a subset of the set of pages that would be in memory with $N + 1$ frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames increases then these n pages will still be the most recently referenced and so, will still be in the memory. While in FIFO, if a page named b came into physical memory before a page – a then priority of replacement of b is greater than that of a , but this is not independent of the number of page frames and hence, FIFO does not follow a stack page replacement policy and therefore suffers Belady's Anomaly.

Example: Consider the following diagram to understand the behavior of a stack-based page replacement algorithm



The diagram illustrates that given the set of pages i.e. $\{0, 1, 2\}$ in 3 frames of memory is not a subset of the pages in memory – $\{0, 1, 4, 5\}$ with 4 frames and it is a violation in the property of stack-based algorithms. This situation can be frequently seen in FIFO algorithm.

Belady's Anomaly in FIFO –

Assuming a system that has no pages loaded in the memory and uses the FIFO Page replacement algorithm. Consider the following reference string:

Case-1: If the system has 3 frames, the given reference string using FIFO page replacement algorithm yields a total of 9 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

1	1	1	2	3	4	1	1	1	2	5	5
2	2	2	3	4	1	2	2	2	5	3	3
		3	4	1	2	5	5	5	3	4	4
PF	X	X	PF	PF	X						

Case-2: If the system has 4 frames, the given reference string using the FIFO page replacement algorithm yields a total of 10 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

1	1	1	1	1	1	2	3	4	5	1	2
2	2	2	2	2	3	4	5	1	2	3	
		3	3	3	4	5	1	2	3	4	
			4	4	4	5	1	2	3	4	5
PF	PF	PF	PF	X	X	PF	PF	PF	PF	PF	PF

It can be seen from the above example that on increasing the number of frames while using the FIFO page replacement algorithm, the number of **page faults increased** from 9 to 10.

Why Stack based algorithms do not suffer Anomaly –

All the stack based algorithms never suffer Belady Anomaly because these type of algorithms assigns a priority to a page (for replacement) that is independent of the number of page frames. Examples of such policies are Optimal, LRU and LFU. Additionally these algorithms also have a good property for simulation, i.e. the miss (or hit) ratio can be computed for any number of page frames with a single pass through the reference string.

In LRU algorithm every time a page is referenced it is moved at the top of the stack, so, the top n pages of the stack are the n most recently used pages. Even if the number of frames is incremented to $n+1$, top of the stack will have $n+1$ most recently used pages. Similar example can be used to calculate the number of page faults in LRU algorithm. Assuming a system that has no pages loaded in the memory and uses the LRU Page replacement algorithm. Consider the following reference string:

Case-1: If the system has 3 frames, the given reference string using the LRU page replacement algorithm yields a total of 10 page faults. The diagram below illustrates the pattern of the page faults occurring in the example.

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3
PF	X	X	PF	PF	PF						

Case-2: If the system has 4 frames, the given reference string on using LRU page replacement algorithm, then total 8 page faults occur. The diagram shows the pattern of the page faults in the example.

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3
			1	2	3	4	4	4	5	1	2
PF	PF	PF	PF	X	X	PF	X	X	PF	PF	PF

How Can Belady's Anomaly Be Removed?

A stack-based approach can be used to get rid of Belady's Algorithm. These are some examples of such algorithms:

- Optimal Page Replacement Algorithm
- Least Recently Used Algorithm (LRU)

These algorithms are based on the idea that if a page is inactive for a long time, it is not being utilised frequently. Therefore, it would be best to forget about this page. This allows for improvised memory management and the abolition of Belady's anomaly.

Conclusion : Various factors substantially affect the number of page faults, such as reference string length and the number of free page frames available. Anomalies also occur due to the small cache size as well as the reckless rate of change of the contents of the cache. Also, the situation of a fixed number of page faults even after increasing the number of frames can also be seen as an anomaly. Often algorithms like **Random page replacement algorithm** are also susceptible to Belady's Anomaly, because it **may behave like first in first out (FIFO)** page replacement algorithm. But Stack based algorithms are generally immune to all such situations as they are guaranteed to give better page hits when the frames are incremented.

Features of Belady's Anomaly:

Page fault rate: Page fault rate is the number of page faults that occur during the execution of a process. Belady's Anomaly occurs when the page fault rate increases as the number of page frames allocated to a process increases.

Page replacement algorithm: Belady's Anomaly is specific to some page replacement algorithms, including the First-In-First-Out (FIFO) algorithm and the Second-Chance algorithm.

System workload: Belady's Anomaly can occur when the system workload changes. Specifically, it can happen when the number of page references in the workload increases.

Page frame allocation: Belady's Anomaly can occur when the page frames allocated to a process are increased, but the total number of page frames in the system remains constant. This is because increasing the number of page frames allocated to a process reduces the number of page faults initially, but when the workload increases, the increased number of page frames can cause the process to evict pages from its working set more frequently, leading to more page faults.

Impact on performance: Belady's Anomaly can significantly impact system performance, as it can result in a higher number of page faults and slower overall system performance. It can also make it challenging to choose an optimal number of page frames for a process.

Advantages :

- **Better insight into algorithm behavior:** Belady's Anomaly can provide insight into how a page replacement algorithm works and how it can behave in different scenarios. This can be helpful in designing and optimizing algorithms for specific use cases.
- **Improved algorithm performance:** In some cases, increasing the number of frames allocated to a process can actually improve algorithm performance, even if it results in more page faults. This is because having more frames can reduce the frequency of page replacement, which can improve overall performance.

Disadvantages :

- **Poor predictability:** Belady's Anomaly can make it difficult to predict how an algorithm will perform with different configurations of frames and pages, which can lead to unpredictable performance and system instability.
- **Increased overhead:** In some cases, increasing the number of frames allocated to a process can result in increased overhead and resource usage, which can negatively impact system performance.
- **Unintuitive behavior:** Belady's Anomaly can result in unintuitive behavior, where increasing the number of frames allocated to a process results in more page faults, which can be confusing for users and system administrators.
- **Difficulty in optimization:** Belady's Anomaly can make it difficult to optimize page replacement algorithms for specific use cases, as the behavior of the algorithm can be unpredictable and inconsistent.

The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be [preemptive or non-preemptive](#).

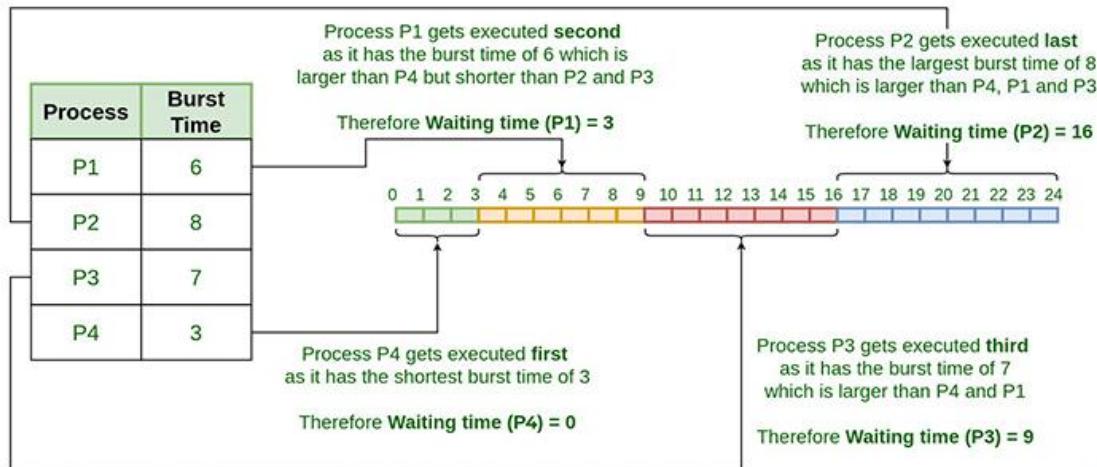
Characteristics of SJF Scheduling:

- Shortest Job first has the advantage of having a minimum average waiting time among all [scheduling algorithms](#).
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
- SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

Shortest Job First (SJF) Scheduling Algorithm



How to compute below times in SJF using a program?

- Completion Time: Time at which process completes its execution.
- Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time
- Waiting Time(W.T): Time Difference between turn around time and burst time.
Waiting Time = Turn Around Time – Burst Time

Program for Non-Preemptive Shortest Job First CPU Scheduling

Non-Preemptive Shortest Job First algorithm can be implemented using [Segment Trees](#) data structure. For detailed implementation of Non-Preemptive **Shortest Job First** scheduling algorithm, please refer: Program for Non-Preemptive Shortest Job First CPU Scheduling.

In this post, we have assumed arrival times as 0, so turn around and completion times are same.

Examples to show working of Non-Preemptive Shortest Job First CPU Scheduling Algorithm:

Example-1: Consider the following table of arrival time and burst time for five processes **P1, P2, P3, P4** and **P5**.

Process	Burst Time	Arrival Time
P1	6 ms	2 ms
P2	2 ms	5 ms
P3	8 ms	1 ms
P4	3 ms	0 ms
P5	4 ms	4 ms

The Shortest Job First CPU Scheduling Algorithm will work on the basis of steps as mentioned below:

At time = 0,

- Process P4 arrives and starts executing

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
0-1ms	P4	0ms		1ms	3ms	2ms

At time= 1,

- Process P3 arrives.
- But, as P4 still needs 2 execution units to complete.
- Thus, P3 will wait till P4 gets executed.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
1-2ms	P4	0ms		1ms	2ms	1ms
	P3	1ms	P3	0ms	8ms	8ms

At time =2,

- Process P1 arrives and is added to the waiting table
- P4 will continue its execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
2-3ms	P4	0ms		1ms	1ms	0ms
	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3, P1	0ms	6ms	6ms

At time = 3,

- Process P4 will finish its execution.
- Then, the burst time of P3 and P1 is compared.
- Process P1 is executed because its burst time is less as compared to P3.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	6ms	5ms

At time = 4,

- Process P5 arrives and is added to the waiting Table.
- P1 will continue execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
4-5ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	5ms	4ms
	P5	4ms	P3, P5	0ms	4ms	4ms

At time = 5,

- Process P2 arrives and is added to the waiting Table.
- P1 will continue execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
5-6ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	4ms	3ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms

At time = 6,

- Process P1 will finish its execution.
- The burst time of P3, P5, and P2 is compared.
- Process P2 is executed because its burst time is the lowest among all.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
6-9ms	P3	1ms	P3	0ms	8ms	8ms
	P4	2ms	P3	3ms	3ms	0ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms

At time=9,

- Process P2 is executing and P3 and P5 are in the waiting Table.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
9-11ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5	2ms	2ms	0ms

At time = 11,

- The execution of Process P2 will be done.
- The burst time of P3 and P5 is compared.
- Process P5 is executed because its burst time is lower than P3.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
11-15ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3	4ms	4ms	0ms

At time = 15,

- Process P5 will finish its execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
15-23ms	P3	4ms		8ms	8ms	0ms

At time = 23,

- Process P3 will finish its execution.
- The overall execution of the processes will be as shown below:

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
0-1ms	P4	0ms		1ms	3ms	2ms
1-2ms	P4	0ms		1ms	2ms	1ms
	P3	1ms	P3	0ms	8ms	8ms
2-3ms	P4	0ms		1ms	1ms	0ms
	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3, P1	0ms	6ms	6ms
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	6ms	5ms
4-5ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	1ms	5ms	4ms
	P5	4ms	P3, P5	0ms	4ms	4ms
5-6ms	P3	1ms	P3	0ms	8ms	8ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
6-9ms	P1	2ms	P3	1ms	4ms	3ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms
	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms	P3	3ms	3ms	0ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5, P2	0ms	2ms	2ms
	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5	2ms	2ms	0ms
9-11ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3, P5	0ms	4ms	4ms
	P2	5ms	P3, P5	2ms	2ms	0ms
11-15ms	P3	1ms	P3	0ms	8ms	8ms
	P5	4ms	P3	4ms	4ms	0ms
15-23ms	P3	4ms		8ms	8ms	0ms

Gantt chart for above execution:



Now, let's calculate the average waiting time for above example:

Advantages of SJF:

- SJF is better than the [First come first serve](#)(FCFS) algorithm as it reduces the average waiting time.
- SJF is generally used for long term scheduling
- It is suitable for the jobs running in batches, where run times are already known.
- SJF is probably optimal in terms of average turnaround time.

Disadvantages of SJF:

- SJF may cause very long turn-around times or [starvation](#).
- In SJF job completion time must be known earlier, but sometimes it is hard to predict.
- Sometimes, it is complicated to predict the length of the upcoming CPU request.
- It leads to the starvation that does not reduce average turnaround time.
- In the **Shortest Remaining Time First (SRTF) scheduling algorithm**, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a **smaller amount of time**.
- [Examples to show working of Preemptive Shortest Job First CPU Scheduling Algorithm:](#)
- **Example-1:** Consider the following table of arrival time and burst time for five processes **P1, P2, P3, P4** and **P5**.

Process	Burst Time	Arrival Time
P1	6 ms	2 ms
P2	2 ms	5 ms

Process	Burst Time	Arrival Time
P3	8 ms	1 ms
P4	3 ms	0 ms
P5	4 ms	4 ms

The Shortest Job First CPU Scheduling Algorithm will work on the basis of steps as mentioned below:

At time = 0,

- Process P4 arrives and starts executing

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
0-1ms	P4	0ms		1ms	3ms	2ms

At time= 1,

- Process P3 arrives.
- But, as P4 has a shorter burst time. It will continue execution.
- Thus, P3 will wait till P4 gets executed.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
1-2ms	P4	0ms	P3	1ms	2ms	1ms
	P3	1ms		0ms	8ms	8ms

At time =2,

- Process P1 arrives with burst time = 6
- As the burst time of P1 is more than that of P4
- Thus, P4 will continue its execution.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
2-3ms	P4	0ms	P3, P1	1ms	1ms	0ms
	P3	1ms		0ms	8ms	8ms
	P1	2ms		0ms	6ms	6ms

At time = 3,

- Process P4 will finish its execution.
- Then, the burst time of P3 and P1 is compared.
- Process P1 is executed because its burst time is less as compared to P3.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms		1ms	6ms	5ms

At time = 4,

- Process P5 arrives.
- Then the burst time of P3, P5, and P1 is compared.
- Process P5 gets executed first among them because its burst time is lowest, and process P1 is **preempted**.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
4-5ms	P3	1ms	P3, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		1ms	4ms	3ms

At time = 5,

- Process P2 arrives.
- The burst time of all processes are compared,

- Process P2 gets executed as its burst time is lowest among all.
- Process P5 is preempted.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
5-6ms	P3	1ms	P3, P5, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		0ms	3ms	3ms
	P2	5ms		1ms	2ms	1ms

At time = 6,

- Process P2 will keep executing.
- It will execute till time = 8 as the burst time of P2 is 2ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
6-7ms	P3	1ms	P3, P5, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		0ms	3ms	3ms
	P2	5ms		1ms	1ms	0ms

At time=7,

- The process P2 finishes its execution.
- Then again the burst time of all remaining processes is compared.
- The Process P5 gets executed because its burst time is lesser than the others.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
7-10ms	P3	1ms	P3, P1	0ms	8ms	8ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
	P1	2ms		0ms	5ms	5ms
	P5	4ms		3ms	3ms	0ms

At time = 10,

- The process P5 will finish its execution.
- Then the burst time of the remaining processes P1 and P3 is compared.
- Thus, process P1 is executed as its burst time is less than P3

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
10-15ms	P3	1ms	P3	0ms	8ms	8ms
	P1	4ms		4ms	5ms	0ms

At time = 15,

- The process P1 finishes its execution and P3 is the only process left.
- P3 will start executing.

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
15-23ms	P3	4ms		8ms	8ms	0ms

At time = 23,

- Process P3 will finish its execution.
- The overall execution of the processes will be as shown below:

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
0-1ms	P4	0ms		1ms	3ms	2ms
1-2ms	P4	0ms	P3	1ms	2ms	1ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
2-3ms	P3	1ms	P3, P1	0ms	8ms	8ms
	P4	0ms		1ms	1ms	0ms
	P3	1ms		0ms	8ms	8ms
	P1	2ms		0ms	6ms	6ms
3-4ms	P3	1ms	P3	0ms	8ms	8ms
	P1	2ms		1ms	6ms	5ms
4-5ms	P3	1ms	P3, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		1ms	4ms	3ms
5-6ms	P3	1ms	P3, P5, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		0ms	3ms	3ms
	P2	5ms		1ms	2ms	1ms
6-7ms	P3	1ms	P3, P5, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		0ms	3ms	3ms
	P2	5ms		1ms	1ms	0ms

Time Instance	Process	Arrival Time	Waiting Table	Execution Time	Initial Burst Time	Remaining Burst Time
7-10ms	P3	1ms	P3, P1	0ms	8ms	8ms
	P1	2ms		0ms	5ms	5ms
	P5	4ms		3ms	3ms	0ms
10-15ms	P3	1ms	P3	0ms	8ms	8ms
	P1	4ms		4ms	5ms	0ms
15-23ms	P3	1ms		8ms	8ms	0ms

Gantt chart for above execution:



Gantt chart for SRTF

Now, lets calculate average [waiting time and turn around time](#):
As we know,

- Turn Around time = Completion time – arrival time
- Waiting Time = Turn around time – burst time

Process	Completion Time	Turn Around Time	Waiting Time
P1	15	15-2 = 13	13-6 = 7
P2	7	7-5 = 2	2-2 = 0
P3	23	23-1 = 22	22-8 = 14
P4	3	3-0 = 3	3-3 = 0
P5	10	10-4 = 6	6-4 = 2

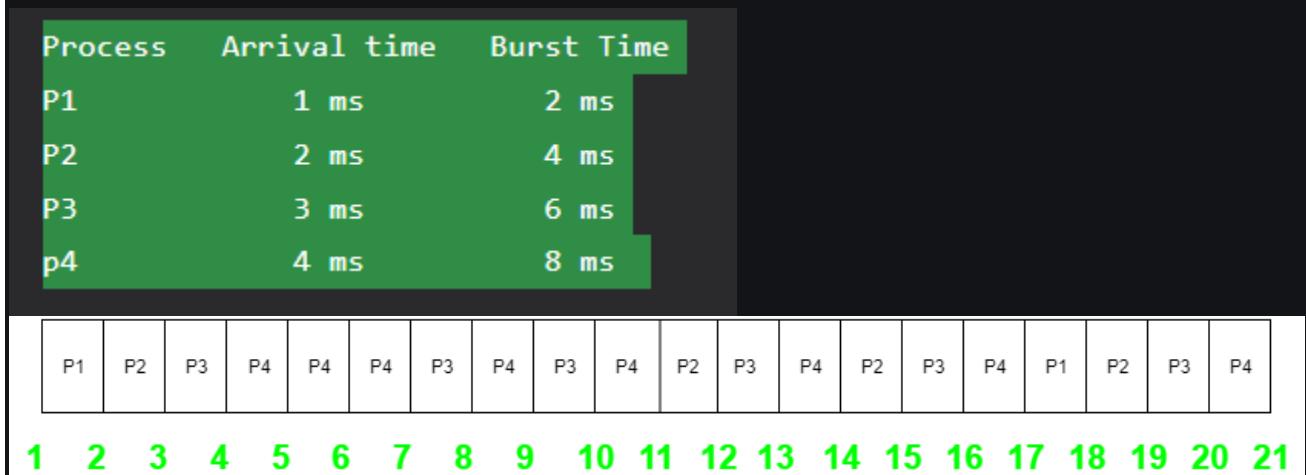
Now,

- **Average Turn around time** = $(13 + 2 + 22 + 3 + 6)/5 = 9.2$
- **Average waiting time** = $(7 + 0 + 14 + 0 + 2)/5 = 23/5 = 4.6$
- Longest Remaining Time First (LRTF) CPU Scheduling Program

LRTF is a preemptive scheduling algorithm. Its tie-breaker is FCFS and if FCFS does not break the tie then, we use process id as the tie-breaker.

Example: Consider the following table of arrival time and burst time for four processes P1, P2, P3, and P4.

Gantt chart will be as following below,



Since completion time (CT) can be directly determined by Gantt chart, and

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive version of First come First Serve CPU Scheduling algorithm.

- Round Robin CPU Algorithm generally focuses on Time Sharing technique.
- The period of time for which a process or job is allowed to run in a pre-emptive method is called time **quantum**.
- Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **end** else the process will go back to the **waiting table** and wait for its next turn to complete the execution.

2.24 Characteristics of Round Robin CPU Scheduling Algorithm

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in [CPU scheduling](#) as a core.
- It is [preemptive](#) as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

2.25 Advantages of Round Robin CPU Scheduling Algorithm

- There is fairness since every process gets equal share of CPU.
- The newly created process is added to end of ready queue.
- A round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum.
- While performing a round-robin scheduling, a particular time quantum is allotted to different jobs.
- Each process gets a chance to reschedule after a particular quantum time in this scheduling.

2.26 Disadvantages of Round Robin CPU Scheduling Algorithm

- There is Larger waiting time and Response time.
- There is Low throughput.
- There is Context Switches.
- Gantt chart seems to come too big (if quantum time is less for scheduling. For Example: 1 ms for big scheduling.)
- Time consuming scheduling for small quantum.

2.27 Examples to show working of Round Robin Scheduling Algorithm

Example-1: Consider the following table of arrival time and burst time for four processes **P1, P2, P3, and P4** and given **Time Quantum = 2**

Process	Burst Time	Arrival Time
P1	5 ms	0 ms
P2	4 ms	1 ms
P3	2 ms	2 ms
P4	1 ms	4 ms

The Round Robin CPU Scheduling Algorithm will work on the basis of steps as mentioned below:

At time = 0,

- The execution begins with process P1, which has burst time 5.
- Here, every process executes for 2 milliseconds (**Time Quantum Period**). P2 and P3 are still in the waiting queue.

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
0-2ms	P1	0ms	P2, P3	P1	2ms	5ms	3ms

At time = 2,

- The processes P2 and P3 arrives in the ready queue and P2 starts executing for **TQ** period

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
2-4ms	P1	0ms	P3, P1	P2	0ms	3ms	3ms
	P2	1ms			2ms	4ms	2ms

At time = 4,

- The process P4 arrives in the **ready queue**,
- Then P3 executes for **TQ** period.

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
4-6ms	P1	0ms	P1, P4, P2	P3	0ms	3ms	3ms
	P2	1ms			0ms	2ms	2ms
	P3	2ms			2ms	2ms	0ms

At time = 6,

- Process P3 completes its execution
- Process P1 starts executing for **TQ** period as it is next in the b.

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
6-8ms	P1	0ms	P4, P2	P1	2ms	3ms	1ms
	P2	1ms			0ms	2ms	2ms

At time = 8,

- Process P4 starts executing, it will not execute for **Time Quantum period** as it has burst time = 1
- Hence, it will execute for only 1ms.

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
8-9ms	P1	0ms	P2, P1	P4	0ms	3ms	1ms
	P2	1ms			0ms	2ms	2ms
	P4	4ms			1ms	1ms	0ms

At time = 9,

- Process P4 completes its execution
- Process P2 starts executing for **TQ** period as it is next in the **ready queue**

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
9-11ms	P1	0ms	P1	P2	0ms	3ms	1ms
	P2	1ms			2ms	2ms	0ms

At time = 11,

- Process P2 completes its execution.
- Process P1 starts executing, it will execute for 1ms only

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
11-12ms	P1	0ms		P1	1ms	1ms	0ms

At time = 12,

- Process P1 completes its execution.
- The overall execution of the processes will be as shown below:

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
0-2ms	P1	0ms	P2, P3	P1	2ms	5ms	3ms
2-4ms	P1	0ms	P3, P1	P2	0ms	3ms	3ms
	P2	1ms			2ms	4ms	2ms
	P1	0ms	P1, P4, P2	P3	0ms	3ms	3ms
4-6ms	P2	1ms		P3	0ms	2ms	2ms
	P3	2ms			2ms	2ms	0ms
6-8ms	P1	0ms	P4, P2	P1	2ms	3ms	1ms
	P2	1ms			0ms	2ms	2ms
	P1	0ms			0ms	3ms	1ms
8-9ms	P2	1ms	P2, P1	P4	0ms	2ms	2ms
	P4	4ms			1ms	1ms	0ms

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
9-11ms	P1	0ms	P1	P2	0ms	3ms	1ms
	P2	1ms			2ms	2ms	0ms
11-12ms	P1	0ms		P1	1ms	1ms	0ms

Gantt chart will be as following below:



Gantt chart for Round Robin Scheduling Algorithm

How to compute below times in Round Robin using a program?

- **Completion Time:** Time at which process completes its execution.
- **Turn Around Time:** Time Difference between completion time and arrival time. **Turn Around Time = Completion Time – Arrival Time**
- **Waiting Time(W.T):** Time Difference between turn around time and burst time. **Waiting Time = Turn Around Time – Burst Time**

Now, let's calculate average waiting time and turn around time:

Processes	AT	BT	CT	TAT	WT
P1	0	5	12	12-0 = 12	12-5 = 7
P2	1	4	11	11-1 = 10	10-4 = 6
P3	2	2	6	6-2 = 4	4-2 = 2
P4	4	1	9	9-4 = 5	5-1 = 4

Now,

We have given some processes with arrival time and Burst Time and we have to find the completion time (CT), Turn Around Time(TAT), Average Turn Around Time (Avg TAT), Waiting Time(WT), Average Waiting Time (AWT) for the given processes.

Prerequisite: [CPU Scheduling | Longest Remaining Time First \(LRTF\) algorithm](#)

LRTF is a preemptive scheduling algorithm. Its tie-breaker is FCFS and if FCFS does not break the tie then, we use process id as the tie-breaker.

Example: Consider the following table of arrival time and burst time for four processes P1, P2, P3, and P4.

Process	Arrival time	Burst Time
P1	1 ms	2 ms
P2	2 ms	4 ms
P3	3 ms	6 ms
P4	4 ms	8 ms

Gantt chart will be as following below,

In the traditional Round Robin scheduling algorithm, all processes were treated equally for processing. The objective of the **Selfish Round Robin** is to give better service to processes that have been executing for a while than to newcomers. It's a more logical and superior implementation compared to the normal Round Robin algorithm.

What is the Selfish Round Robin scheduling algorithm?

The Selfish Round Robin (SRR) scheduling algorithm is a modification of the traditional Round Robin algorithm. The objective of SRR is to give better service to processes that have been executing for a while than to newcomers. In SRR, processes in the ready list are partitioned into two lists: NEW and ACCEPTED. The New processes wait while Accepted processes are serviced by the Round Robin. The priority of a new process increases at the rate 'a' while the priority of an accepted process increases at the rate 'b'.

How does SRR scheduling ensure fairness among processes?

A: SRR scheduling ensures fairness among processes by providing each process with an equal amount of CPU time. This is achieved by using a Round Robin approach, where each process is given a time quantum to execute. Once a process completes its time quantum, it is placed at the end of the queue, and the next process is given a chance to execute.

Is SRR scheduling suitable for real-time systems?

A: No, SRR scheduling is not suitable for real-time systems that require strict timing constraints. In such systems, it is essential to ensure that high-priority processes receive CPU time when they need it, which is not possible with SRR scheduling.

Implementation:

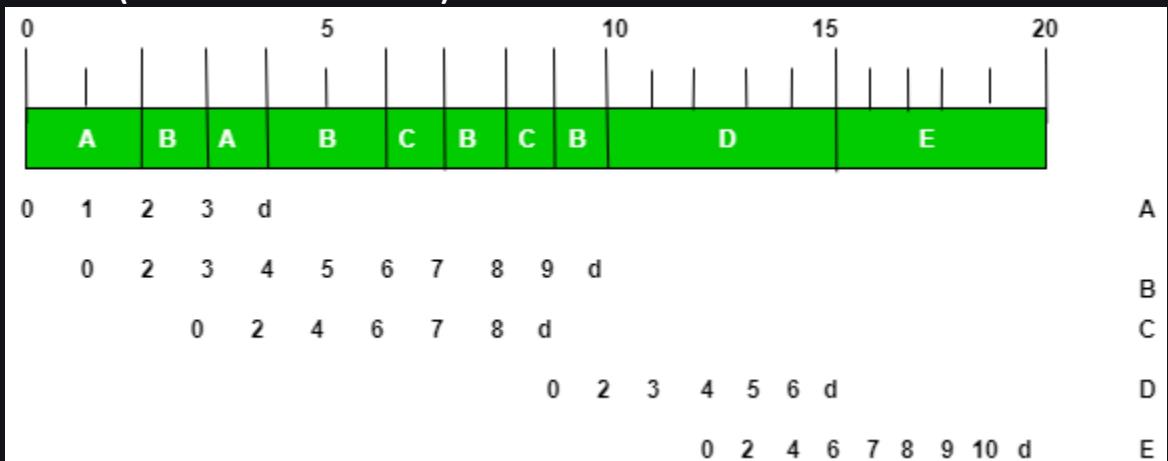
- Processes in the ready list are partitioned into two lists: NEW and ACCEPTED.
- The New processes wait while Accepted processes are serviced by the Round Robin.

- Priority of a new process increases at rate 'a' while the priority of an accepted process increases at rate 'b'.
- When the priority of a new process reaches the priority of an accepted process, that new process becomes accepted.
- If all accepted processes finish, the highest priority new process is accepted.
- **STEP 1:** Assume that initially there are no ready processes, when the first one, A, arrives. It has priority 0, to begin with. Since there are no other accepted processes, A is accepted immediately.
- STEP 2:** After a while, another process, B, arrives. As long as $b / a < 1$, B's priority will eventually catch up to A's, so it is accepted; now both A and B have the same priority.
- STEP 3:** All accepted processes share a common priority (which rises at rate b); that makes this policy easy to implement i.e any new process's priority is bound to get accepted at some point. So no process has to experience starvation.
- STEP 4:** Even if $b / a > 1$, A will eventually finish, and then B can be accepted.

- Example on Selfish Round Robin –

Process name	Arrival time	Service required	Finish time
A	0	3	4
B	1	5	10
C	3	2	9
D	9	5	15
E	12	5	20

- Solution (where $a = 2$ and $b = 1$) –



- Explanation:

- Process A gets accepted as soon as it comes at time $t = 0$. So its priority is increased only by 'b' i.e '1' after each second. B enters at time $t = 1$ and goes to the waiting queue. So its priority gets increased by 'a' i.e. '2' at time $t = 2$. At this point priority of A = priority of B = 2.
- So now both processes A & B are in the accepted queue and are executed in a round-robin fashion. At time $t = 3$ process C enters the waiting queue. At time $t = 6$ the priority of process C catches up to the priority of process B and then they start executing in a Round Robin manner. When B finishes execution at time $t = 10$, D is automatically promoted to the accepted queue.
- Similarly, when D finishes execution at time $t = 15$, E is automatically promoted to the accepted queue.
- Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. The process with the highest priority is to be executed first and so on. Processes with the same priority are executed on a first-come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement. Also priority can be decided on the ratio of average I/O to average CPU burst time.
- **Implementation:**

Process	Burst Time	Priority
P1	10	2
P2	5	0
P3	8	1

P1	P3	P2
-----------	-----------	-----------

.

0	10	18	23
---	----	----	----

- **Starvation** or indefinite blocking is a phenomenon associated with the Priority scheduling algorithms, in which a process ready for the CPU (resources) can wait to run indefinitely because of low priority. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. There have been rumors that in 1967 Priority Scheduling was used in IBM 7094 at MIT, and they found a low-priority process that had not been submitted till 1973.

Process	Burst time	Priority
1	10	2
2	5	0
3	8	1



- As we see in the above example process having higher priority than other processes getting CPU earlier. We can think of a scenario in which only one process is having very low-priority (for example 127) and we are giving other process with high-priority, this can lead indefinitely waiting for the process for CPU which is having low-priority, this leads to **Starvation**. Further we have also discuss about the solution of starvation.

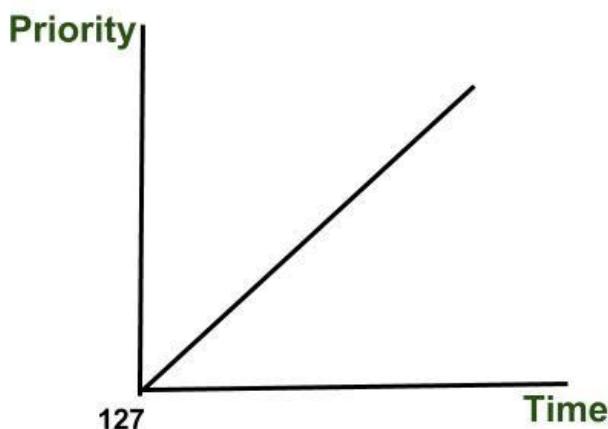
Differences between [Deadlock](#) and Starvation in OS are as follows:

1. Deadlock occurs when none of the processes in the set is able to move ahead due to occupancy of the required resources by some other process as shown in the figure below, on the other hand, Starvation occurs when a process waits for an indefinite period of time to get the resource it requires.
2. Another name for deadlock is **Circular Waiting**. Another name for starvation is **Lived lock**.

3. When deadlock occurs no process can make progress, while in starvation apart from the victim process other processes can progress or proceed.

Solution to Starvation: Aging

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes. Eventually, even a process with an initial priority of 127 would take no more than 32 hours for the priority 127 process to age to a priority-0 process.



Limitations of the aging technique are listed below:

- **Increased complexity:** Aging requires additional overhead to periodically adjust the priorities of waiting processes, which can increase the overall complexity of the scheduling algorithm.
- **Overhead:** Frequent priority adjustments may introduce additional overhead, which can reduce the overall efficiency of the scheduling algorithm.
- **Unpredictable behavior:** Aging can lead to unpredictable behavior if the aging rate is not set appropriately. If the aging rate is too slow, it may take a long time for low-priority processes to receive the required resources. On the other hand, if the aging rate is too fast, it can cause high-priority processes to starve.
- **Unfairness:** Aging can also be unfair to newly arrived processes as it prioritizes long waiting processes over new ones. This can lead to a situation where new processes are starved of resources while long waiting processes continue to receive them.

Given N processes with their [Arrival times and Burst times](#), the task is to find average waiting time and an average turn around time using HRRN scheduling algorithm.

The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

HRRN CPU Scheduling:

- **Highest Response Ratio Next** is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithm.
- The **criteria** for HRRN is **Response Ratio**, and the **mode** is **Non-Preemptive**.
- HRRN is basically considered as the modification of [Shortest Job First](#) in order to reduce the problem of [starvation](#).
- In comparison with SJF, during HRRN scheduling algorithm, the CPU is allotted to the next process which has the **highest response ratio** and not to the process having less burst time.

Here, **W** is the **waiting time** of the process so far and **S** is the **Burst time** of the process.

Advantages of HRRN CPU Scheduling

- HRRN Scheduling algorithm generally gives better performance than the [shortest job first](#) Scheduling.
- There is a reduction in waiting time for longer jobs and also it encourages shorter jobs.

Disadvantages of HRRN CPU Scheduling

- The on ground implementation of HRRN scheduling is not possible as it is not possible know the burst time of every job in advance.
- In this scheduling, there may occur overload on the CPU.

Performance of HRRN –

- Shorter Processes are favoured.
- Aging without service increases ratio, longer jobs can get past shorter jobs.

Let us consider the following examples.

Example-1: Consider the following table of arrival time and burst time for four processes P1, P2, P3, P4 and P5.

Processes	Arrival time	Burst Time
P1	0ms	3ms
P2	2ms	6ms
P3	4ms	4ms
P4	6ms	5ms

Processes	Arrival time	Burst Time
P5	8ms	2ms

The Highest Response Ratio Next CPU Scheduling Algorithm will work on the basis of steps as mentioned below:

At time= 0,

- Available Processes: P1, Hence P1 starts executing till its completion.

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
0-2ms	P1	0ms		P1	2ms	3ms	1ms

At time = 2,

- Available Processes: P1, P2
- But P1 keep executing as HRRN is a non-preemptive algorithm and thus it will finish its execution

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
2-3ms	P1	0ms	P2	P1	1ms	4ms	0ms
	P2	2ms			0ms	6ms	6ms

At time = 3,

- Process P1 finish its execution
- Process P2 starts executing as it is only process available.

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
3-4ms	P2	2ms		P2	1ms	6ms	5ms

At time = 4,

- Process P3 arrives and wait for process P2 to execute

- Process P2 continue its execution

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
4-6ms	P2	2ms	P3	P2	2ms	5ms	3ms
	P3	4ms			0ms	4ms	4ms

At time = 6,

- Process P4 arrives and wait for the process P2 to execute

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
6-8ms	P2	2ms	P3, P4	P2	2ms	3ms	1ms
	P3	4ms			0ms	4ms	4ms
	P4	6ms			0ms	5ms	5ms

At time = 8

- Process P5 arrives and wait for its execution in the ready queue

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
8-9ms	P2	2ms	P3, P4, P5	P2	1ms	4ms	0ms
	P3	4ms			0ms	4ms	4ms
	P4	6ms			0ms	5ms	5ms
	P5	8ms			0ms	2ms	2ms

At time = 9,

- Process P2 completes its execution
- Now there are 3 processes available, P3, P4 and P5. Since, P3, P4, P5 were available after 4, 6 and 8 units respectively.
- Therefore, waiting time for P3, P4 and P5 are $(9 - 4 =)5$, $(9 - 6 =)3$, and $(9 - 8 =)1$ unit respectively.
- Using the formula given above, $(\text{Response Ratio} = (\text{W} + \text{S})/\text{S})$ calculate the Response Ratios of P3, P4 and P5 respectively as 2.25, 1.6 and 1.5.
- Clearly, P3 has the highest Response Ratio and so it gets scheduled

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
9-13ms	P3	4ms	P4, P5	P3	4ms	4ms	0ms
	P4	6ms			0ms	5ms	5ms
	P5	8ms			0ms	2ms	2ms

At time = 13,

- Available Processes: P4 and P5
- Response Ratios of P4 and P5 are 2.4 and 3.5 respectively using the above formula.
- Clearly, P5 has the highest Response Ratio and so it gets scheduled

Time Instance	Process	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
13-15ms	P4	6ms	P4	P5	0ms	5ms	5ms
	P5	8ms			2ms	2ms	0ms

At time = 15,

- After completion of process P5, Process P4 is selected at last and execute till it gets finished

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
15-20ms	P4	6ms		P4	5ms	5ms	0ms

At time = 20,

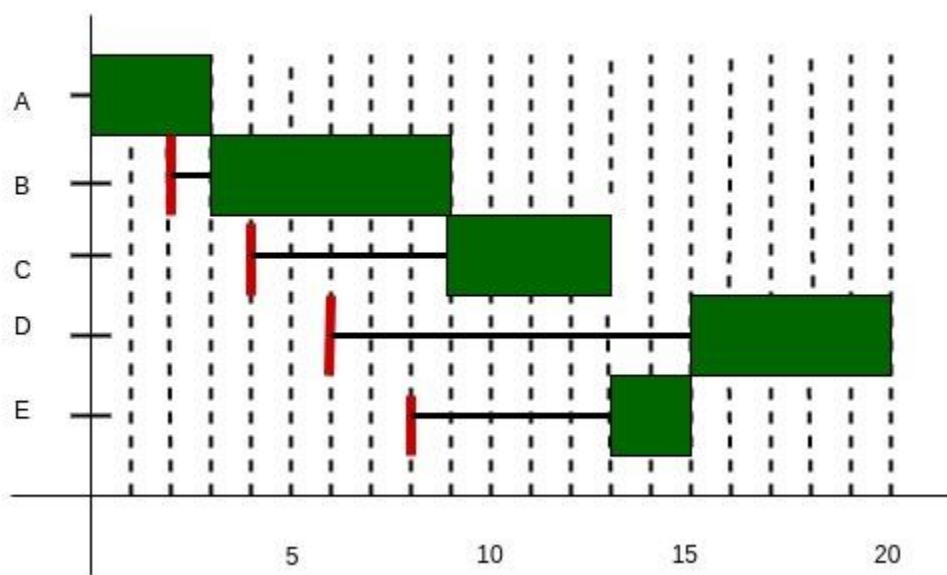
- Process P4 will finish its execution.
- The overall execution of the processes will be as shown below:

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
0-2ms	P1	0ms		P1	2ms	3ms	1ms
2-3ms	P1	0ms	P2	P1	1ms	4ms	0ms
	P2	2ms			0ms	6ms	6ms
3-4ms	P2	2ms		P2	1ms	6ms	5ms
4-6ms	P2	2ms	P3	P2	2ms	5ms	3ms
	P3	4ms			0ms	4ms	4ms
6-8ms	P2	2ms	P3, P4	P2	2ms	3ms	1ms
	P3	4ms			0ms	4ms	4ms
	P4	6ms			0ms	5ms	5ms
8-9ms	P2	2ms	P3, P4, P5	P2	1ms	4ms	0ms
	P3	4ms			0ms	4ms	4ms

Time Instance	Processes	Arrival Time	Ready Queue	Running Queue	Execution Time	Initial Burst Time	Remaining Burst Time
	P4	6ms			0ms	5ms	5ms
	P5	8ms			0ms	2ms	2ms
9-13ms	P3	4ms	P4, P5	P3	4ms	4ms	0ms
	P4	6ms			0ms	5ms	5ms
	P5	8ms			0ms	2ms	2ms
	P4	6ms			0ms	5ms	5ms
13-15ms	P5	8ms		P5	2ms	2ms	0ms
	P4	6ms			5ms	5ms	0ms

Gantt Chart –

Gantt Chart -



Since, completion time (C.T) can be directly determined by Gantt chart, and

Therefore, final table look like,

Processes	AT	BT	CT	TAT	WT
P1	0	3	3	$3-0 = 3$	$3-3 = 0$
P2	2	6	9	$9-2 = 7$	$7-6 = 1$
P3	4	4	13	$13-4 = 9$	$9-4 = 5$
P4	6	5	20	$20-6 = 14$	$14-5 = 9$
P5	8	2	15	$15-8 = 7$	$7-2 = 5$

Output:

Implementation of HRRN Scheduling –

- Input the number of processes, their arrival times and burst times.
- Sort them according to their arrival times.
- At any given time calculate the response ratios and select the appropriate process to be scheduled.
- Calculate the turn around time as **completion time – arrival time**.
- Calculate the waiting time as **turn around time – burst time**.
- Turn around time divided by the burst time gives the normalized turn around time.
- Sum up the waiting and turn around times of all processes and divide by the number of processes to get the **average waiting** and **turn around time**.

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process and a **background (batch)** process. These two classes have different scheduling needs. For this kind of situation, **Multilevel Queue Scheduling** is used.

Now, let us see how it works.

Features of Multilevel Queue (MLQ) CPU Scheduling:

- **Multiple queues:** In MLQ scheduling, processes are divided into multiple queues based on their priority, with each queue having a different priority level. Higher-priority processes are placed in queues with higher priority levels, while lower-priority processes are placed in queues with lower priority levels.
- **Priorities assigned:** Priorities are assigned to processes based on their type, characteristics, and importance. For example, interactive processes like user input/output may have a higher priority than batch processes like file backups.
- **Preemption:** Preemption is allowed in MLQ scheduling, which means a higher priority process can preempt a lower priority process, and the CPU is allocated to the higher priority process. This helps ensure that high-priority processes are executed in a timely manner.
- **Scheduling algorithm:** Different scheduling algorithms can be used for each queue, depending on the requirements of the processes in that queue. For example, Round Robin scheduling may be used for interactive processes, while First Come First Serve scheduling may be used for batch processes.
- **Feedback mechanism:** A feedback mechanism can be implemented to adjust the priority of a process based on its behavior over time. For example, if an interactive process has been waiting in a lower-priority queue for a long time, its priority may be increased to ensure it is executed in a timely manner.
- **Efficient allocation of CPU time:** MLQ scheduling ensures that processes with higher priority levels are executed in a timely manner, while still allowing lower priority processes to execute when the CPU is idle.
- **Fairness:** MLQ scheduling provides a fair allocation of CPU time to different types of processes, based on their priority and requirements.
- **Customizable:** MLQ scheduling can be customized to meet the specific requirements of different types of processes.

2.28 Advantages of Multilevel Queue CPU Scheduling:

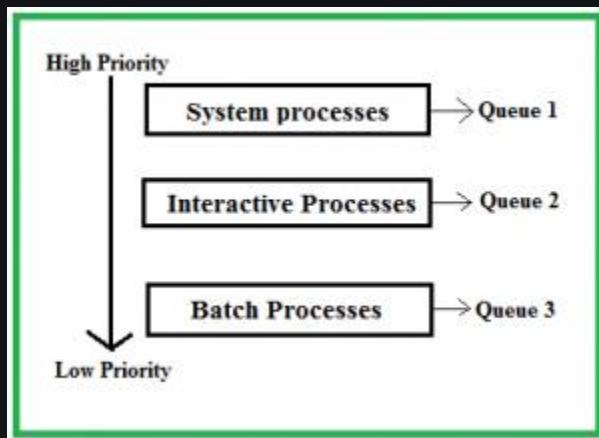
- **Low scheduling overhead:** Since processes are permanently assigned to their respective queues, the overhead of scheduling is low, as the scheduler only needs to select the appropriate queue for execution.
- **Efficient allocation of CPU time:** The scheduling algorithm ensures that processes with higher priority levels are executed in a timely manner, while still allowing lower priority processes to execute when the CPU is idle. This ensures optimal utilization of CPU time.
- **Fairness:** The scheduling algorithm provides a fair allocation of CPU time to different types of processes, based on their priority and requirements.
- **Customizable:** The scheduling algorithm can be customized to meet the specific requirements of different types of processes. Different scheduling algorithms can be used for each queue, depending on the requirements of the processes in that queue.

- **Prioritization:** Priorities are assigned to processes based on their type, characteristics, and importance, which ensures that important processes are executed in a timely manner.
- **Preemption:** Preemption is allowed in Multilevel Queue Scheduling, which means that higher-priority processes can preempt lower-priority processes, and the CPU is allocated to the higher-priority process. This helps ensure that high-priority processes are executed in a timely manner.

2.29 Disadvantages of Multilevel Queue CPU Scheduling:

- Some processes may starve for CPU if some higher priority queues are never becoming empty.
- It is inflexible in nature.
- There may be added complexity in implementing and maintaining multiple queues and scheduling algorithms.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of processes System processes, Interactive processes, and Batch Processes. All three processes have their own queue. Now, look at the below figure.



The Description of the processes in the above diagram is as follows:

- **System Processes:** The CPU itself has its own process to run which is generally termed a System Process.
- **Interactive Processes:** An Interactive Process is a type of process in which there should be the same type of interaction.
- **Batch Processes:** Batch processing is generally a technique in the Operating system that collects the programs and data together in the form of a **batch** before the **processing** starts.

All three different type of processes have their own queue. Each queue has its own Scheduling algorithm. For example, queue 1 and queue 2 use **Round Robin** while queue 3 can use **FCFS** to schedule their processes.

Scheduling among the queues: What will happen if all the queues have some processes? Which process should get the CPU? To determine this Scheduling among the queues is necessary. There are two ways to do so –

- Fixed priority preemptive scheduling method** – Each queue has absolute priority over the lower priority queue. Let us consider the following priority order **queue 1 > queue 2 > queue 3**. According to this algorithm, no process in the batch queue(queue 3) can run unless queues 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
- Time slicing** – In this method, each queue gets a certain portion of CPU time and can use it to schedule its own processes. For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

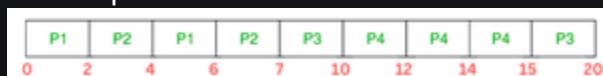
Example Problem:

Consider the below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the **Gantt chart** of the problem:



Working:

- At starting, both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round-robin fashion and completes after 7 units
- Then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 seconds and
- After its completion P3 takes the CPU and completes its execution.

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like [Multilevel Queue\(MLQ\) Scheduling](#) but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics of Multilevel Feedback Queue Scheduling:

- In a [multilevel queue-scheduling](#) algorithm, processes are permanently assigned to a queue on entry to the system, and processes are allowed to move between queues.
- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,

Features of Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling:

Multiple queues: Similar to MLQ scheduling, MLFQ scheduling divides processes into multiple queues based on their priority levels. However, unlike MLQ scheduling, processes can move between queues based on their behavior and needs.

Priorities adjusted dynamically: The priority of a process can be adjusted dynamically based on its behavior, such as how much CPU time it has used or how often it has been blocked. Higher-priority processes are given more CPU time and lower-priority processes are given less.

Time-slicing: Each queue is assigned a time quantum or time slice, which determines how much CPU time a process in that queue is allowed to use before it is preempted and moved to a lower priority queue.

Feedback mechanism: MLFQ scheduling uses a feedback mechanism to adjust the priority of a process based on its behavior over time. For example, if a process in a lower-priority queue uses up its time slice, it may be moved to a higher-priority queue to ensure it gets more CPU time.

Preemption: Preemption is allowed in MLFQ scheduling, meaning that a higher-priority process can preempt a lower-priority process to ensure it gets the CPU time it needs.

Advantages of Multilevel Feedback Queue Scheduling:

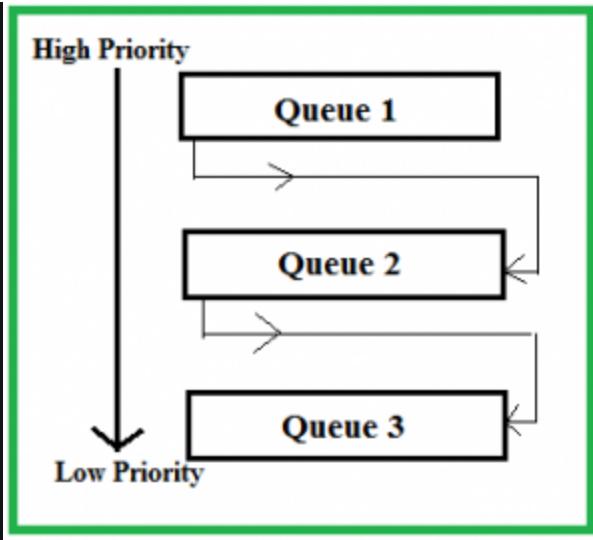
- It is more flexible.
- It allows different processes to move between different queues.
- It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.

Disadvantages of Multilevel Feedback Queue Scheduling:

- The selection of the best scheduler, it requires some other means to select the values.
- It produces more CPU overheads.
- It is the most complex algorithm.

Multilevel feedback queue scheduling, however, allows a process to move between queues. Multilevel Feedback Queue Scheduling (**MLFQ**) keeps analyzing the behavior (time of execution) of processes and according to which it changes its priority.

Now, look at the diagram and explanation below to understand it properly.



Now let us suppose that queues 1 and 2 follow [round robin](#) with time quantum 4 and 8 respectively and queue 3 follow [FCFS](#).

Implementation of MFQS is given below –

- When a process starts executing the operating system can insert it into any of the above three queues depending upon its **priority**. For example, if it is some background process, then the operating system would not like it to be given to higher priority queues such as queues 1 and 2. It will directly assign it to a lower priority queue i.e. queue 3. Let's say our current process for consideration is of significant priority so it will be given **queue 1**.
- In queue 1 process executes for 4 units and if it completes in these 4 units or it gives CPU for I/O operation in these 4 units then the priority of this process does not change and if it again comes in the ready queue then it again starts its execution in Queue 1.
- If a process in queue 1 does not complete in 4 units then its priority gets reduced and it is shifted to queue 2.
- Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 units. In a general case if a process does not complete in a time quantum then it is shifted to the lower priority queue.
- In the last queue, processes are scheduled in an FCFS manner.
- A process in a lower priority queue can only execute only when higher priority queues are empty.
- A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

What is the need for such complex Scheduling?

- Firstly, it is more flexible than multilevel queue scheduling.
- To optimize turnaround time algorithms like SJF are needed which require the running time of processes to schedule them. But the running time of the process is not known in advance. MFQS runs a process for a time quantum and then it can change its priority(if it is a long process). Thus it learns from past behavior of the process and then predicts its future behavior. This way it tries to run a shorter process first thus optimizing turnaround time.
- MFQS also reduces the response time.

Example: Consider a system that has a CPU-bound process, which requires a burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum '2' seconds and in each level it is incremented by '5' seconds. Then how many times the process will be interrupted and in which queue the process will terminate the execution?

Solution:

- Process P needs 40 Seconds for total execution.
- At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.
- At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.
- At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.
- At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.
- At Queue 5 it executes for 2 seconds and then it completes.
- Hence the process is interrupted 4 times and completed on queue 5.
- **Lottery Scheduling** is a type of process scheduling, somewhat different from other Scheduling. Processes are scheduled in a random manner. Lottery scheduling can be **preemptive or non-preemptive**. It also solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has a non-zero probability of being selected at each scheduling operation. In this scheduling every process has some tickets and the scheduler picks a random ticket and process having that ticket is the winner and it is executed for a time slice and then another ticket is picked by the scheduler. These tickets represent the share of processes. A process having a higher number of tickets give it more chance to get chosen for execution.
- **Example –** If we have two processes A and B having 60 and 40 tickets respectively out of total of 100 tickets. CPU share of A is 60% and that of B is 40%. These shares are calculated probabilistically and not deterministically.
- **Explanation:**

1. We have two processes A and B. A has 60 tickets (ticket number 1 to 60) and B has 40 tickets (ticket no. 61 to 100).
2. Scheduler picks a random number from 1 to 100. If the picked no. is from 1 to 60 then A is executed otherwise B is executed.
3. An example of 10 tickets picked by the Scheduler may look like this follows:

1. A is executed 7 times and B is executed 3 times. As you can see that A takes 70% of the CPU and B takes 30% which is not the same as what we need as we need A to have 60% of the CPU and B should have 40% of the CPU. This happens because shares are calculated probabilistically but in a long run(i.e when no. of tickets picked is more than 100 or 1000) we can achieve a share percentage of approx. 60 and 40 for A and B respectively.

Ways to manipulate tickets –

- **Ticket Currency** – Scheduler gives a certain number of tickets to different users in a currency and users can give it to their processes in a different currency. E.g. Two users A and B are given 100 and 200 tickets respectively. User A is running two processes and gives 50 tickets to each in A's own currency. B is running 1 process and gives it all 200 tickets in B's currency. Now at the time of scheduling tickets of each process are converted into global currency i.e A's process will have 50 tickets each and B's process will have 200 tickets and scheduling is done on this basis.
- **Transfer Tickets** – A process can pass its tickets to another process.
- **Ticket inflation** – With this technique a process can temporarily raise or lower the number of tickets it owns.

Advantages of Lottery Process Scheduling:

1. **Fairness:** Lottery process scheduling is designed to be fair because every process has a chance of being selected. This ensures that no process is starved of CPU time, and it also helps to prevent monopolization of the CPU by a single process.
2. **Flexibility:** The lottery process scheduling algorithm is highly flexible and can be easily adapted to handle various scheduling requirements, such as real-time scheduling, multi-processor scheduling, and so on.
3. **Simple implementation:** The implementation of the lottery process scheduling algorithm is relatively simple compared to other scheduling algorithms, making it easier to develop and maintain.
4. **Combining with other scheduling algorithms:** Lottery scheduling can be combined with other scheduling algorithms such as round-robin or priority scheduling to achieve specific scheduling goals.

Disadvantages of Lottery Process Scheduling:

1. **Overhead:** The lottery process scheduling algorithm requires additional overhead because it must generate lottery tickets for each process and randomly select a winner. This overhead can be significant in systems with many processes.
2. **Complexity:** Although the implementation of the lottery process scheduling algorithm is relatively simple, the algorithm itself can be quite complex, especially when compared to other scheduling algorithms such as round-robin or FIFO.
3. **Security:** Lottery process scheduling is not a secure algorithm because it is based on randomness, which can be manipulated. An attacker could potentially gain an unfair advantage by manipulating the generation of lottery tickets, leading to security vulnerabilities.
4. In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.
5. **Why is multiple-processor scheduling important?**
Multiple-processor scheduling is important because it enables a computer system to perform multiple tasks simultaneously, which can greatly improve overall system performance and efficiency.
6. **How does multiple-processor scheduling work?**
Multiple-processor scheduling works by dividing tasks among multiple processors in a computer system, which allows tasks to be processed simultaneously and reduces the overall time needed to complete them.
7. Approaches to Multiple-Processor Scheduling –
8. One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**. A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity –

Processor Affinity means a process has an **affinity** for the processor on which it is currently running. When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**. There are two types of processor affinity:

1. **Soft Affinity** – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity** – Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like *sched_setaffinity()* that supports hard affinity.

Load Balancing –

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of processes which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processes awaiting the CPU. There are two general approaches to load balancing :

1. **Push Migration** – In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration** – Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors –

In multicore processors **multiple processor** cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the

operating system as a separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip. However multicore processors may **complicate** the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread. There are two ways to multithread a processor :

1. **Coarse-Grained Multithreading** – In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.
2. **Fine-Grained Multithreading** – This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

Virtualization and Threading –

In this type of **multiple-processor** scheduling even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU to each of virtual machines running on the system and then schedules the use of physical CPU among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines. Each virtual machine has a guest operating system installed and applications run within that guest. Each guest operating system may be assigned for specific use cases, applications or users including time sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. A time sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in a very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that

individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's. **Virtualizations** can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines. **Reference – [Operating System Principles – Galvin](#)**

3. Process and Threads

Within a program, a **Thread** is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if CPU is more than 1 otherwise two threads have to context switch for that single CPU.

3.1 Why Do We Need Thread?

- Threads run in parallel improving the application performance. Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use interprocess communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own [Thread Control Block \(TCB\)](#). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB). As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

3.2 Why Multi-Threading?

A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below.

Multithreading is a technique used in operating systems to improve the performance and responsiveness of computer systems. Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices.

3.3 Difference Between Process and Thread

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result, threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

3.4 Advantages of Thread

- **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Faster context switch:** Context switch time between threads is lower compared to the process context switch. Process context switching requires more overhead from the CPU.
- **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
- **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process. Note: Stacks and registers can't be shared among the threads. Each thread has its own stack and registers.
- **Communication:** Communication between multiple threads is easier, as the threads share a common address space. while in the process we have to follow some specific communication techniques for communication between the two processes.
- **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

3.5 Types of Threads

Threads are of two types. These are described below.

- [User Level Thread](#)
- [Kernel Level Thread](#)

User Level Thread and Kernel Level Thread

User Level Threads

User Level Thread is a type of thread that is not created using system calls. The kernel has no work in the management of user-level threads. User-level threads can be easily implemented by the user. In case when user-level threads are single-handed processes, kernel-level thread manages them. Let's look at the advantages and disadvantages of User-Level Thread.

Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.
- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

Disadvantages of User-Level Threads

- There is a lack of coordination between Thread and Kernel.
- In case of a page fault, the whole process can be blocked.

Kernel Level Threads

A kernel Level Thread is a type of thread that can recognize the Operating system easily. Kernel Level Threads has its own thread table where it keeps track of the system. The operating System Kernel helps in managing threads. Kernel Threads have somehow longer context switching time. Kernel helps in the management of threads.

Advantages of Kernel-Level Threads

- It has up-to-date information on all threads.
- Applications that block frequency are to be handled by the Kernel-Level Threads.
- Whenever any process requires more time to process, Kernel-Level Thread provides more time to it.

Disadvantages of Kernel-Level threads

- Kernel-Level Thread is slower than User-Level Thread.
- Implementation of this type of thread is a little more complex than a user-level thread.

For more, refer to the [Difference Between User-Level Thread and Kernel-Level Thread](#).

3.6 Components of Threads

These are the basic components of the Operating System.

- Stack Space
- Register Set
- Program Counter

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter
2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources etc.

Similarity between Threads and Processes –

- Only one thread or process is active at a time
- Within process both execute sequential
- Both can create children
- Both can be scheduled by the operating system: Both threads and processes can be scheduled by the operating system to execute on the CPU. The operating system is responsible for assigning CPU time to the threads and processes based on various scheduling algorithms.
- Both have their own execution context: Each thread and process has its own execution context, which includes its own register set, program counter, and stack. This allows each thread or process to execute independently and make progress without interfering with other threads or processes.
- Both can communicate with each other: Threads and processes can communicate with each other using various inter-process communication (IPC) mechanisms such as shared memory, message queues, and pipes. This allows threads and processes to share data and coordinate their activities.
- Both can be preempted: Threads and processes can be preempted by the operating system, which means that their execution can be interrupted at any time. This allows the operating system to switch to another thread or process that needs to execute.
- Both can be terminated: Threads and processes can be terminated by the operating system or by other threads or processes. When a thread or process is terminated, all of its resources, including its execution context, are freed up and made available to other threads or processes.

Differences between Threads and Processes –

- **Resources:** Processes have their own address space and resources, such as memory and file handles, whereas threads share memory and resources with the program that created them.
- **Scheduling:** Processes are scheduled to use the processor by the operating system, whereas threads are scheduled to use the processor by the operating system or the program itself.
- **Creation:** The operating system creates and manages processes, whereas the program or the operating system creates and manages threads.
- **Communication:** Because processes are isolated from one another and must rely on inter-process communication mechanisms, they generally have more difficulty communicating with one another than threads do. Threads, on the other hand, can interact with other threads within the same programme directly.

Threads, in general, are lighter than processes and are better suited for concurrent execution within a single programme. Processes are commonly used to run separate programmes or to isolate resources between programmes.

Types of Threads:

1. **User Level thread (ULT)** – Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.
 - **Advantages of ULT –**
 - Can be implemented on an OS that doesn't support multithreading.
 - Simple representation since thread has only program counter, register set, stack space.
 - Simple to create since no intervention of kernel.
 - Thread switching is fast since no OS calls need to be made.
 - **Limitations of ULT –**
 - No or less co-ordination among the threads and Kernel.
 - If one thread causes a page fault, the entire process blocks.
2. **Kernel Level Thread (KLT)** – Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.
 - **Advantages of KLT –**
 - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
 - Good for applications that frequently block.

- **Limitations of KLT –**

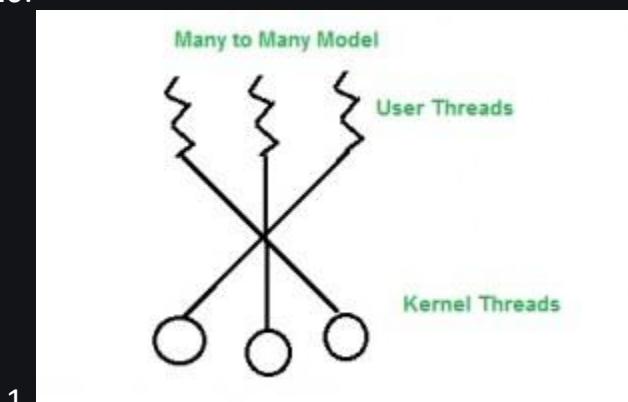
- Slow and inefficient.
- It requires thread control block so it is an overhead.

Summary:

1. Each ULT has a process that keeps track of the thread using the Thread table.
2. Each KLT has Thread Table (TCB) as well as the Process Table (PCB).
3. Many operating systems support kernel thread and user thread in a combined way. Example of such system is Solaris. Multi threading model are of three types.

7. Many to Many Model

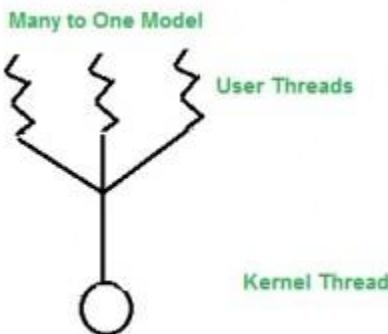
8. In this model, we have multiple user threads multiplex to same or lesser number of kernel level threads. Number of kernel level threads are specific to the machine, advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn't block if a particular thread is blocked.
9. It is the best multi threading model.
- 10.



11.

12. Many to One Model

13. In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time.
14. The thread management is done on the user level so it is more efficient.
- 15.



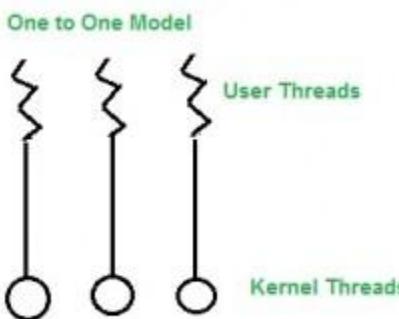
16.

17. One to One Model

In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on multiple processor. Problem with this model is that creating a user thread requires the corresponding kernel thread.

18. As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.

19.



20.

21. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Zombie state: When a process is created in UNIX using fork() system call, the parent process is cloned. If the parent process calls wait() system call, then the execution of the parent is suspended until the child is terminated. At the termination of the child, a 'SIGCHLD' signal is generated by the kernel which is delivered to the parent. Parent, on receipt of 'SIGCHLD' reads the status of the child from the process table. Even though the child is terminated, there is an entry in the process table corresponding to the child where the status is stored. When the parent collects the status, this entry is deleted (or we also say the parent reaped its child process) . Thus, all the traces of the child process are removed from the system. If the parent decides not to wait for the child's termination and executes its subsequent task, then at the termination of the child, the exit status is not read. Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.

• C

```
// A C program to demonstrate working of
// fork() and process table entries.

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

int main()
{
    int i;
    int pid = fork();

    if (pid == 0)
    {
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        printf("I am Parent\n");
        while(1);
    }
}
```

Output :

Now check the process table using the following command in the terminal

\$ ps -eaf

Learn more

Pause
Unmute

```
maverick@maverick-Inspiron-5548: ~          ↑ 5.5 KIB/s En (95%) Sun Apr 2 3:57 AM C: Kishlay
maverick 2193 1581 0 03:32 ?    00:00:00 /bin/sh -c /usr/lib/x86_64-linux-gnu/zeitgeist/zeitgeist-maybe-vacuum; /usr/bin/zeitgeist-daemon
maverick 2197 2193 0 03:32 ?    00:00:00 /usr/bin/zeitgeist-daemon
maverick 2212 1581 0 03:32 ?    00:00:00 /usr/lib/x86_64-linux-gnu/zeitgeist-fs
maverick 2231 1581 0 03:32 ?    00:00:00 /usr/lib/gvfs/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec_spaw/6
maverick 2292 1809 0 03:32 ?    00:00:00 update-notifier
maverick 2327 1581 0 03:33 ?    00:00:00 /usr/lib/gvfs/gvfsd-metadata
maverick 2332 1809 0 03:33 ?    00:00:00 /usr/lib/x86_64-linux-gnu/deja-dup/deja-dup-monitor
maverick 2343 1581 5 03:34 ?    00:01:21 /opt/google/chrome/chrome
maverick 2348 2343 0 03:34 ?    00:00:00 cat
maverick 2349 2343 0 03:34 ?    00:00:00 cat
maverick 2352 2343 0 03:34 ?    00:00:00 /opt/google/chrome/chrome --type=zygote --enable-crash-reporter=460786ff-7d77-47c5-8c1e-ee77e8b3df89,
maverick 2353 2352 0 03:34 ?    00:00:00 /opt/google/chrome-nacl_helper
maverick 2356 2352 0 03:34 ?    00:00:00 /opt/google/chrome/chrome --type=zygote --enable-crash-reporter=460786ff-7d77-47c5-8c1e-ee77e8b3df89,
maverick 2414 2343 3 03:34 ?    00:00:45 /opt/google/chrome/chrome --type=gpu-process --field-trial-handle=1 --enable-crash-reporter=460786ff-7
maverick 2416 2414 0 03:34 ?    00:00:00 /opt/google/chrome/chrome --type=gpu-broker
maverick 2426 2356 0 03:34 ?    00:00:07 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=1DCFDEA5553CD
maverick 2428 2356 1 03:34 ?    00:00:21 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=F921AE21B7DCB
maverick 2475 2356 0 03:34 ?    00:00:01 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=A89F37C8F2BF6
maverick 2479 2356 0 03:34 ?    00:00:10 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=7727DB4884093
maverick 2483 2356 0 03:34 ?    00:00:01 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=A88BC34647051
maverick 2487 2356 0 03:34 ?    00:00:00 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=FA04A7005040D
maverick 2492 2356 0 03:34 ?    00:00:02 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=18560D5587ECF
maverick 2808 2356 2 03:35 ?    00:00:38 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=B092BD8061565
root 2825 2 0 03:35 ?    00:00:00 [kworker/3:1]
maverick 3096 2356 14 03:36 ?    00:03:03 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=5EEE7B03F800B
root 3190 2 0 03:38 ?    00:00:00 [kworker/u16:1]
root 3284 2 0 03:41 ?    00:00:00 [kworker/0:2]
maverick 3352 1581 0 03:43 ?    00:00:00 /usr/lib/x86_64-linux-gnu/notify OSD
root 3566 2 0 03:50 ?    00:00:00 [kworker/1:0]
maverick 3593 1581 0 03:51 ?    00:00:00 /usr/lib/gnome-terminal/gnome-terminal-server
maverick 3600 3593 0 03:51 pts/4    00:00:00 bash
root 3672 2 0 03:52 ?    00:00:00 [kworker/u16:0]
root 3673 2 0 03:52 ?    00:00:00 [kworker/0:3]
root 3674 2 0 03:52 ?    00:00:00 [kworker/3:0]
root 3696 2 0 03:54 ?    00:00:00 [kworker/2:0]
maverick 3768 3600 99 03:56 pts/4    00:00:41 ./a.out
maverick 3769 3768 0 03:56 pts/4    00:00:00 [a.out] <defunct>
root 3772 2 0 03:56 ?    00:00:00 [kworker/1:2]
root 3791 2 0 03:57 ?    00:00:00 [kworker/2:3]
maverick 3793 3593 0 03:57 pts/17    00:00:00 bash
root 3809 2 0 03:57 ?    00:00:00 [kworker/3:2]
maverick 3813 3793 0 03:57 pts/17    00:00:00 ps -eaf
maverick@maverick-Inspiron-5548: ~$ █
```

Here the entry **[a.out] defunct** shows the zombie process.

Why do we need to prevent the creation of the Zombie process?

There is one process table per system. The size of the process table is finite. If too many zombie processes are generated, then the process table will be full. That is, the system will not be able to generate any new process, then the system will come to a standstill. Hence, we need to prevent the creation of zombie processes.

Different ways in which the creation of Zombie can be Prevented

1. Using wait() system call: When the parent process calls wait(), after the creation of a child, it indicates that, it will wait for the child to complete and it will reap the exit status of the child. The parent process is suspended(waits in a waiting queue) until the child is terminated. It must be understood that during this period, the parent process does nothing just wait.

- C

```
// A C program to demonstrate working of
```

```

// fork()/wait() and Zombie processes
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();
    if (pid==0)
    {
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        wait(NULL);
        printf("I am Parent\n");
        while(1);
    }
}

```

(Point 2. is wrong and doesn't prevent zombie creation, its the job of parent process to remove the child process from process list and is not done by the kernel, infact SIG_IGN is the default handler for SIGCHLD signal to the parent process).

2. By ignoring the 'SIGCHLD' signal: When a child is terminated, a corresponding SIGCHLD signal is delivered to the parent, if we call the 'signal(SIGCHLD,SIG_IGN)', then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table Thus, no zombie is created. However, in this case, the parent cannot know about the exit status of the child.

- C

```

// A C program to demonstrate ignoring
// SIGCHLD signal to prevent Zombie processes
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;

```

```

int pid = fork();
if (pid == 0)
    for (i=0; i<20; i++)
        printf("I am Child\n");
else
{
    signal(SIGCHLD,SIG_IGN);
    printf("I am Parent\n");
    while(1);
}
}

```

3. By using a signal handler: The parent process installs a signal handler for the SIGCHLD signal. The signal handler calls wait() system call within it. In this scenario, when the child terminated, the SIGCHLD is delivered to the parent. On receipt of SIGCHLD, the corresponding handler is activated, which in turn calls the wait() system call. Hence, the parent collects the exit status almost immediately and the child entry in the process table is cleared. Thus no zombie is created.

```

// A C program to demonstrate handling of
// SIGCHLD signal to prevent Zombie processes.
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

void func(int signum)
{
    wait(NULL);
}

int main()
{
    int i;
    int pid = fork();
    if (pid == 0)
        for (i=0; i<20; i++)
            printf("I am Child\n");
    else
    {
        signal(SIGCHLD, func);
        printf("I am Parent\n");
    }
}

```

```

        while(1);
    }
}

```

Output:

```
maverick@maverick-Inspiron-5548:~ 2.1 KiB/s En (95%) Sun Apr 2 4:11 AM Kishlay
Maverick 2327 1581 0 03:33 ? 00:00:00 /usr/lib/gvfsd-metadata
Maverick 2332 1809 0 03:33 ? 00:00:00 /usr/lib/x86_64-linux-gnu/deja-dup/deja-dup-monitor
Maverick 2343 1581 5 03:34 ? 00:01:53 /opt/google/chrome/chrome
Maverick 2348 2343 0 03:34 ? 00:00:00 cat
Maverick 2349 2343 0 03:34 ? 00:00:00 cat
Maverick 2352 2343 0 03:34 ? 00:00:00 /opt/google/chrome/chrome --type=zygote --enable-crash-reporter=460786ff-7d77-47c5-8c1e-ee77e8b3df89,
Maverick 2353 2352 0 03:34 ? 00:00:00 /opt/google/chrome-nacl_helper
Maverick 2356 2352 0 03:34 ? 00:00:00 /opt/google/chrome/chrome --type=zygote --enable-crash-reporter=460786ff-7d77-47c5-8c1e-ee77e8b3df89,
Maverick 2414 2343 2 03:34 ? 00:01:04 /opt/google/chrome/chrome --type=gpu-process --field-trial-handle=1 --enable-crash-reporter=460786ff-7
Maverick 2416 2414 0 03:34 ? 00:00:00 /opt/google/chrome/chrome --type=gpu-broker
Maverick 2426 2356 0 03:34 ? 00:00:07 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=1DCFDEA5553CD
Maverick 2428 2356 1 03:34 ? 00:00:24 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=F921AE21B7DCB
Maverick 2475 2356 0 03:34 ? 00:00:02 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=A89F37C8F2BF6
Maverick 2479 2356 0 03:34 ? 00:00:14 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=7727DB4884093
Maverick 2483 2356 0 03:34 ? 00:00:02 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=A888C34647051
Maverick 2487 2356 0 03:34 ? 00:00:01 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=FA04A7005040D
Maverick 2492 2356 0 03:34 ? 00:00:02 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=18560D5587ECF
Maverick 2808 2356 3 03:35 ? 00:01:13 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=B092BD8061565
Maverick 3096 2356 12 03:36 ? 00:04:20 /opt/google/chrome/chrome --type=renderer --field-trial-handle=1 --primordial-pipe-token=5EEE7B03F800B
root 3190 2 0 03:38 ? 00:00:00 [kworker/u16:1]
Maverick 3352 1581 0 03:43 ? 00:00:00 /usr/lib/x86_64-linux-gnu/notify-osd
root 3566 2 0 03:50 ? 00:00:00 [kworker/1:0]
Maverick 3593 1581 0 03:51 ? 00:00:01 /usr/lib/gnome-terminal/gnome-terminal-server
Maverick 3600 3593 0 03:51 pts/4 00:00:00 bash
root 3672 2 0 03:52 ? 00:00:00 [kworker/u16:0]
root 3673 2 0 03:52 ? 00:00:00 [kworker/0:3]
root 3696 2 0 03:54 ? 00:00:00 [kworker/2:0]
root 3772 2 0 03:56 ? 00:00:00 [kworker/1:2]
root 3791 2 0 03:57 ? 00:00:00 [kworker/2:3]
root 3809 2 0 03:57 ? 00:00:00 [kworker/3:2]
root 3882 2 0 04:00 ? 00:00:00 [kworker/0:0]
root 3951 2 0 04:02 ? 00:00:00 [kworker/u16:2]
root 3960 2 0 04:02 ? 00:00:00 [kworker/3:0]
root 3995 2 0 04:04 ? 00:00:00 [kworker/2:1]
Maverick 4012 3593 0 04:06 pts/17 00:00:00 bash
root 4024 2 0 04:06 ? 00:00:00 [kworker/0:1]
root 4094 2 0 04:07 ? 00:00:00 [kworker/3:1]
root 4169 2 0 04:10 ? 00:00:00 [kworker/2:2]
root 4170 2 0 04:10 ? 00:00:00 [kworker/2:4]
root 4186 2 0 04:10 ? 00:00:00 [kworker/u16:3]
Maverick 4198 3600 96 04:10 pts/4 00:00:11 ./a.out
Maverick 4207 4012 0 04:10 pts/17 00:00:00 ps -eaf
maverick@maverick-Inspiron-5548:~$
```

Here no any **[a.out] defunct** i.e. no Zombie process is created.

4. Double fork: This involves creating a grandchild process which is then orphaned by the parent process. This ensures that the grandchild process is inherited by the init process, which reaps it automatically. This prevents the creation of a zombie process.

- C

```
// A C program to demonstrate double fork
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int status;
    pid_t pid = fork();

    if (pid == 0) {
        pid_t child_pid = fork();
```

```

    if (child_pid == 0) {
        printf("I am grandchild\n");
        exit(0);
    }
    else {
        printf("I am child\n");
        exit(0);
    }
}
else {
    wait(&status);
    printf("I am parent\n");
    while(1);
}
}

```

5. Using the `waitpid()` system call: This system call is similar to the `wait()` system call, but it allows the parent process to wait for a specific child process to terminate. This way, the parent can collect the exit status of the child and prevent the creation of a zombie process.

- C

```

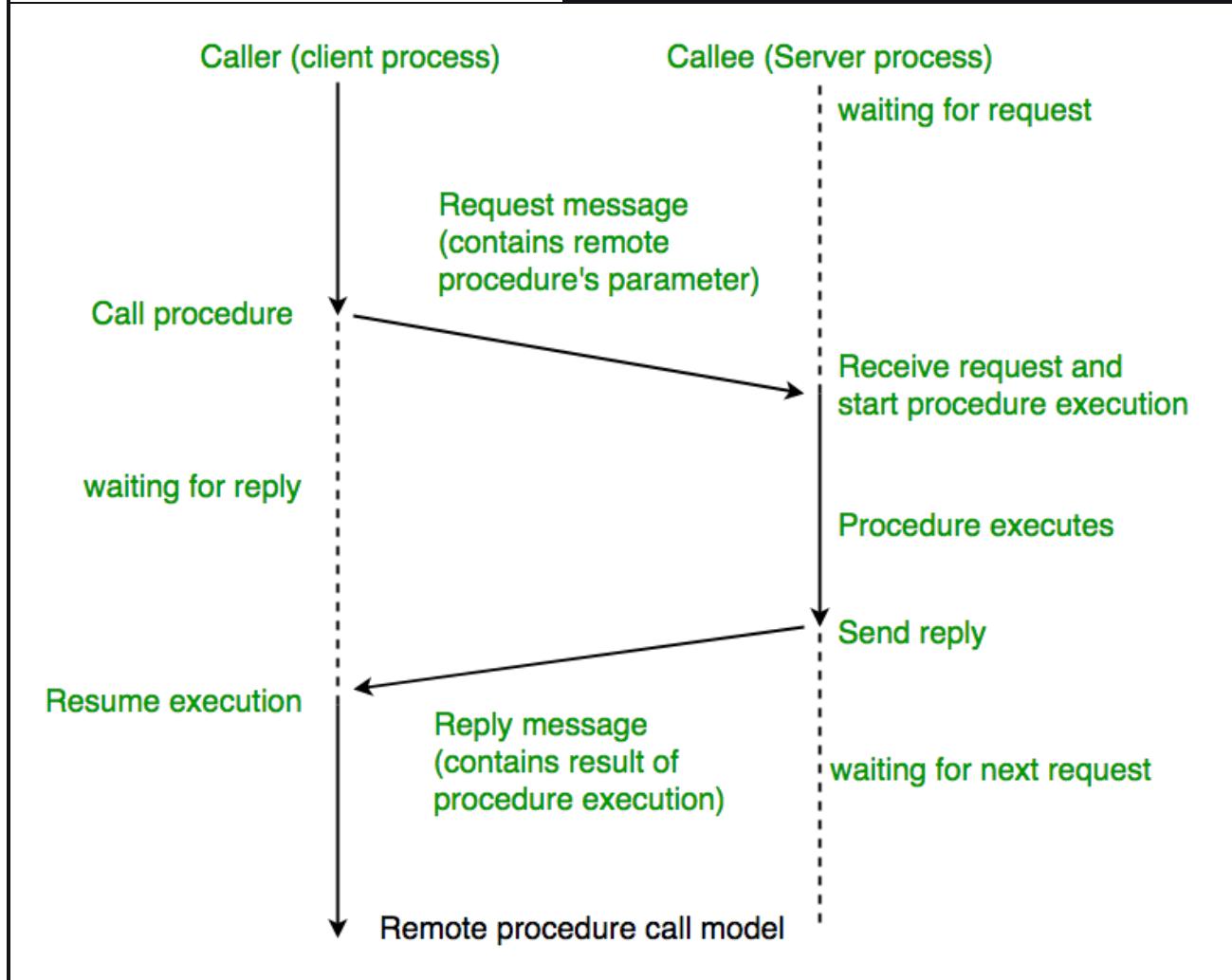
// A C program to demonstrate the use of waitpid()
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>

int main()
{
    int i;
    int pid = fork();
    if (pid == 0){
        for (i=0; i<20; i++)
            printf("I am Child\n");
    }
    else
    {
        int status;
        waitpid(pid, &status, 0);
        printf("I am Parent\n");
        while(1);
    }
}

```

} **Remote Procedure Call (RPC)** is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

When making a Remote Procedure Call:

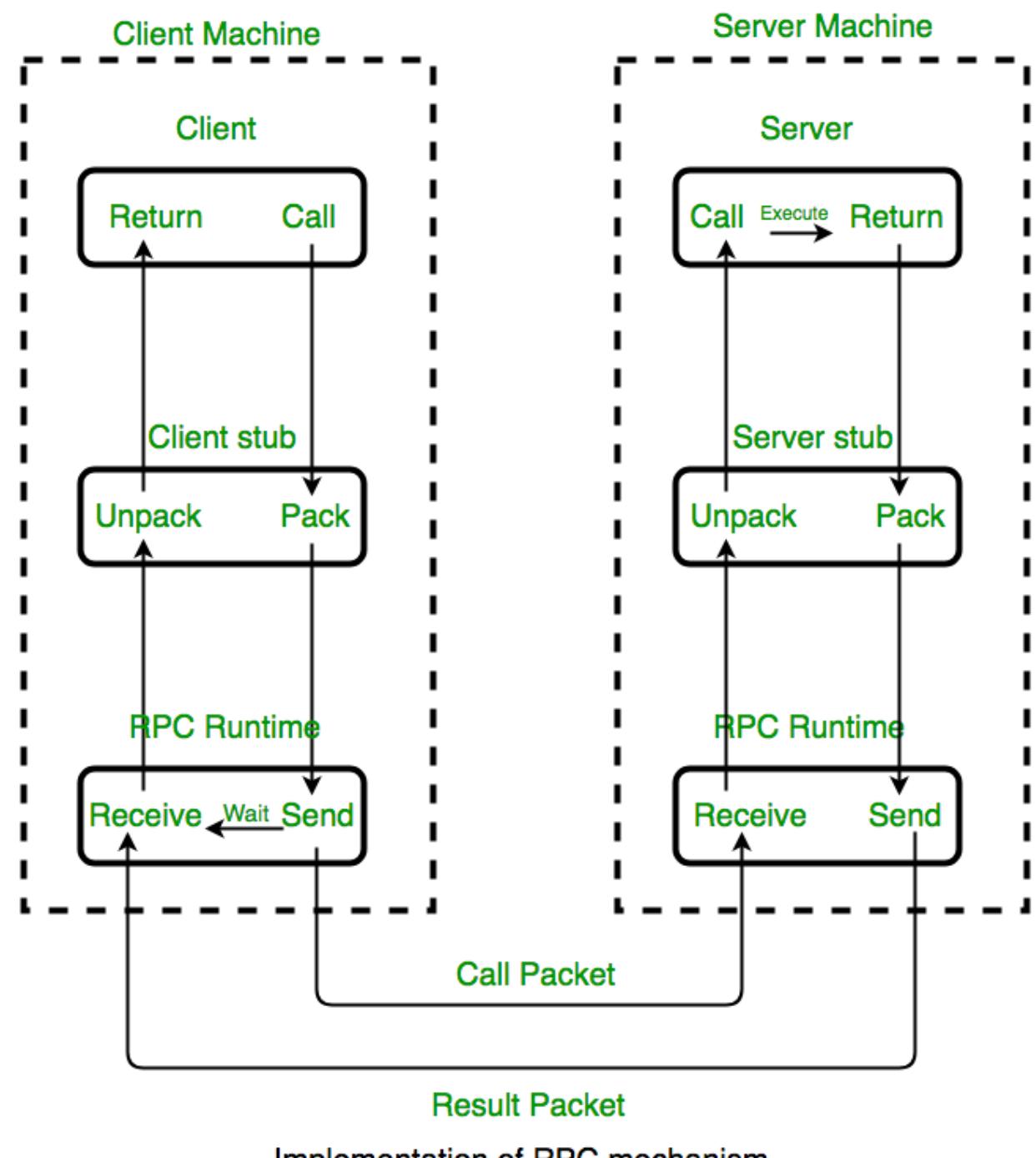


1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

NOTE: **RPC** is especially well suited for client-server (**e.g. query-response**) interaction in which the flow of control **alternates between the caller and callee**. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



The following steps take place during a RPC :

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (**e.g., via a normal procedure call return**), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

Key Considerations for Designing and Implementing RPC Systems are:

- **Security:** Since RPC involves communication over the network, security is a major concern. Measures such as authentication, encryption, and authorization must be implemented to prevent unauthorized access and protect sensitive data.
- **Scalability:** As the number of clients and servers increases, the performance of the RPC system must not degrade. Load balancing techniques and efficient resource utilization are important for scalability.
- **Fault tolerance:** The RPC system should be resilient to network failures, server crashes, and other unexpected events. Measures such as redundancy, failover, and graceful degradation can help ensure fault tolerance.
- **Standardization:** There are several RPC frameworks and protocols available, and it is important to choose a standardized and widely accepted one to ensure interoperability and compatibility across different platforms and programming languages.
- **Performance tuning:** Fine-tuning the RPC system for optimal performance is important. This may involve optimizing the network protocol, minimizing the data transferred over the network, and reducing the latency and overhead associated with RPC calls.

RPC ISSUES :

Issues that must be addressed:

1. RPC Runtime:

RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**

2. Stub:

The function of the stub is to **provide transparency to the programmer-written application code.**

- **On the client side**, the stub handles the interface between the client's local procedure call and the run-time system, marshalling and unmarshalling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.
- **On the server side**, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

3. Binding: How does the client know who to call, and where the service resides?

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Binding consists of two parts:

- Naming:
- Locating:
 1. **A Server** having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.
 2. **A Client** must import an (exported) interface before communication can begin.

4. The call semantics associated with RPC :

It is mainly classified into following choices-

- **Retry request message** –
Whether to retry sending a request message when a server has failed or the receiver didn't receive the message.
- **Duplicate filtering** –
Remove the duplicate server requests.
- **Retransmission of results** –
To resend lost messages without re-executing the operations at the server side.

ADVANTAGES :

1. RPC provides **ABSTRACTION** i.e message-passing nature of network communication is hidden from the user.
2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.
3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.
4. With RPC code re-writing / re-developing effort is minimized.
5. Process-oriented and thread oriented models supported by RPC.

4. Disk Management

A file system is a method an operating system uses to store, organize, and manage files and directories on a storage device. Some common types of file systems include:

1. **FAT (File Allocation Table):** An older [file system](#) used by older versions of Windows and other operating systems.
2. **NTFS (New Technology File System):** A modern file system used by Windows. It supports features such as file and folder permissions, compression, and encryption.
3. **ext (Extended File System):** A file system commonly used on Linux and [Unix-based operating systems](#).
4. **HFS (Hierarchical File System):** A file system used by macOS.
5. **APFS (Apple File System):** A new file system introduced by Apple for their Macs and iOS devices.

The advantages of using a file system

1. **Organization:** A file system allows files to be organized into directories and subdirectories, making it easier to manage and locate files.
2. **Data protection:** File systems often include features such as file and folder permissions, backup and restore, and error detection and correction, to protect data from loss or corruption.
3. **Improved performance:** A well-designed [file system](#) can improve the performance of reading and writing data by organizing it efficiently on disk.

Disadvantages of using a file system

1. **Compatibility issues:** Different file systems may not be compatible with each other, making it difficult to transfer data between different operating systems.
2. **Disk space overhead:** File systems may use some disk space to store metadata and other overhead information, reducing the amount of space available for user data.
3. **Vulnerability:** File systems can be vulnerable to data corruption, malware, and other security threats, which can compromise the stability and security of the system.

A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From the user's perspective, a file is the smallest allotment of logical secondary storage.

The name of the file is divided into two parts as shown below:

- name
- extension, separated by a period.

4.1 The Following Issues Are Handled By The File System

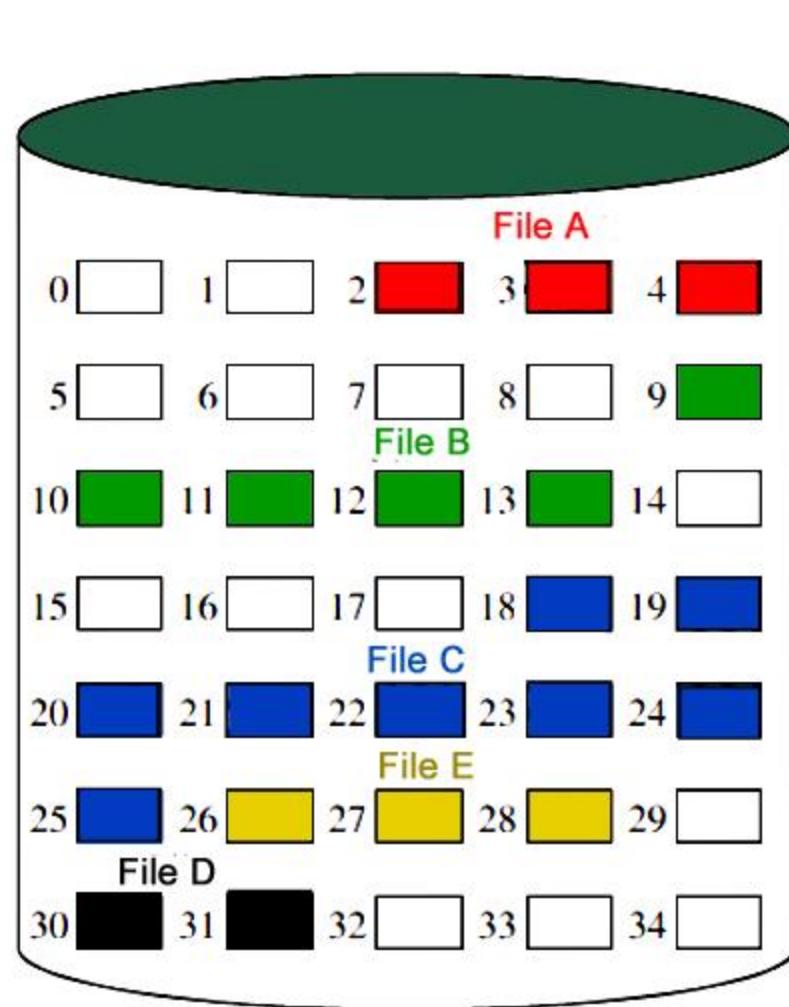
We've seen a variety of data structures where the file could be kept. The [file system](#)'s job is to keep the files organized in the best way possible. A free space is created on the hard drive whenever a file is deleted from it. To reallocate them to other files, many of these spaces may need to be recovered. Choosing where to

store the files on the hard disc is the main issue with files one block may or may not be used to store a file. It may be kept in the disk's non-contiguous blocks. We must keep track of all the blocks where the files are partially located.

4.2 FILE ALLOCATION METHODS

Continuous Allocation

A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply $b+i-1$.



File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Disadvantage

- External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. A compaction algorithm will be necessary to free up additional space on the disk.

- Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

Linked Allocation(Non-contiguous allocation)

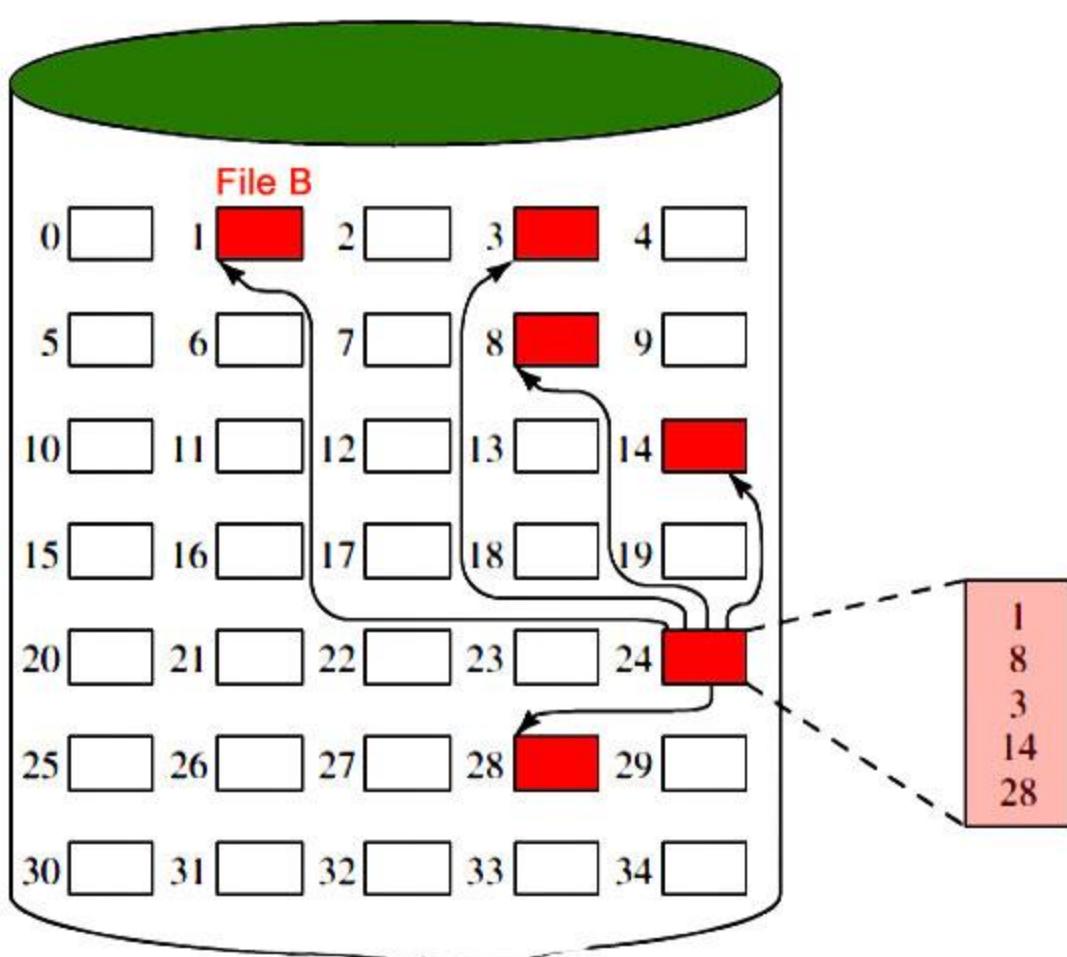
Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. An increase in file size is always possible if a free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of the file.

Disadvantage

- Internal fragmentation exists in the last disk block of the file.
- There is an overhead of maintaining the pointer in every disk block.
- If the pointer of any disk block is lost, the file will be truncated.
- It supports only the sequential access of files.

Indexed Allocation

It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. The allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates [external fragmentation](#), whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.



File allocation table

File name	Index block
• • • File B • • •	• • • 24 • • •

Disk Free Space Management

Just as the space that is allocated to files must be managed, so the space that is not currently allocated to any file must be managed. To perform any of the file allocation techniques, it is necessary to know what blocks on the disk are available. Thus we need a

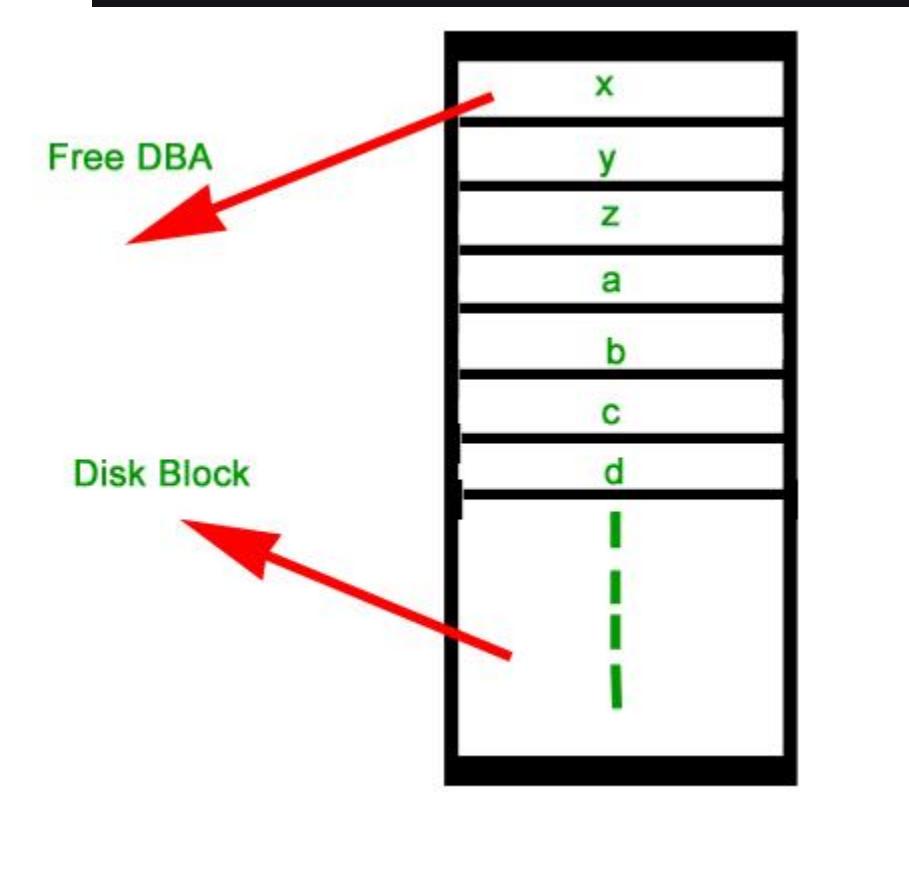
disk allocation table in addition to a file allocation table. The following are the approaches used for free space management.

1. **Bit Tables:** This method uses a vector containing one bit for each block on the disk. Each entry for a 0 corresponds to a free block and each 1 corresponds to a block in use.

For example 00011010111100110001

In this vector every bit corresponds to a particular block and 0 implies that that particular block is free and 1 implies that the block is already occupied. A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks. Thus, a bit table works well with any of the file allocation methods. Another advantage is that it is as small as possible.

2. **Free Block List:** In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved block of the disk.



Unix file system is a logical method of **organizing and storing** large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system. Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file

system is a directory called “root” which is represented by a “/”. All other files are “descendants” of root.

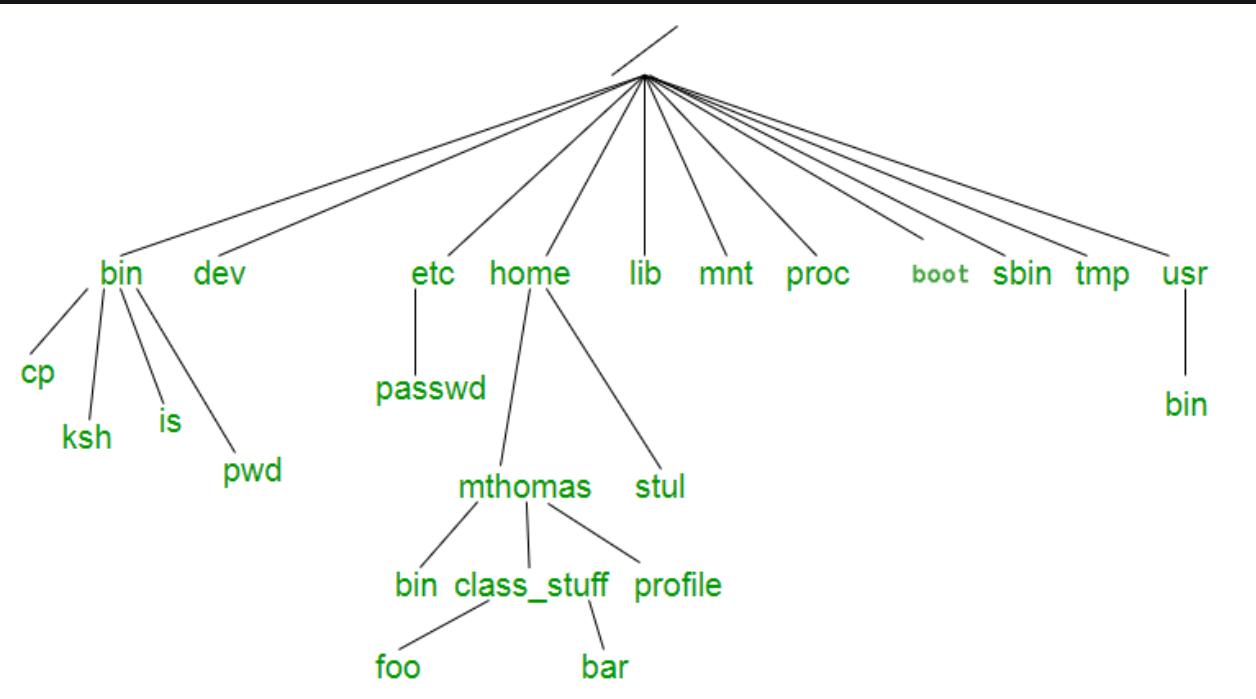
The Unix file system is a hierarchical file system used by Unix-based operating systems to store and organize files and directories. It is a tree-like structure that starts with a single directory called the root directory, which is denoted by a forward slash (/) character.

The Unix file system uses a directory hierarchy that allows for easy navigation and organization of files. Directories can contain both files and other directories, and each file or directory has a unique name.

Unix file system also uses a set of permissions to control access to files and directories. Each file and directory has an owner and a group associated with it, and permissions can be set to allow or restrict access to these entities.

One of the most important features of the Unix file system is its support for symbolic links, which are pointers to other files or directories. This allows for flexible organization of files and directories without having to physically move them around.

Overall, the Unix file system is a robust and flexible system that has been used for decades and continues to be the foundation for many modern operating systems.



Directories or Files and their description –

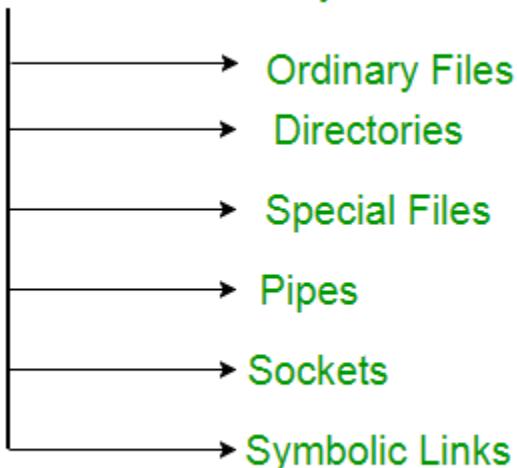
- **/** : The slash / character alone denotes the root of the filesystem tree.
- **/bin** : Stands for “binaries” and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.
- **/boot** : Contains all the files that are required for successful booting process.

- **/dev** : Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices.
- **/etc** : Contains system-wide configuration files and system databases. Originally also contained “dangerous maintenance utilities” such as init, but these have typically been moved to /sbin or elsewhere.
- **/home** : Contains the home directories for the users.
- **/lib** : Contains system libraries, and some critical files such as kernel modules or device drivers.
- **/media** : Default mount point for removable devices, such as USB sticks, media players, etc.
- **/mnt** : Stands for “mount”. Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.
- **/proc** : procfs virtual filesystem showing information about processes as files.
- **/root** : The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.
- **/tmp** : A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
- **/usr** : Originally the directory holding user home directories, its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).
- **/usr/bin** : This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.
- **/usr/include** : Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language.
- **/usr/lib** : Stores the required libraries and data files for programs stored within /usr or elsewhere.
- **/var** : A short for “variable.” A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
- **/var/log** : Contains system log files.

- **/var/mail** : The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.
- **/var/spool** : Spool directory. Contains print jobs, mail spools and other queued tasks.
- **/var/tmp** : A place for temporary files which should be preserved between system reboots.

Types of Unix files – The UNIX files system contains several different types of files :

Classification of Unix File System :



1. Ordinary files – An ordinary file is a file on the system that contains data, text, or program instructions.

- Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
- Always located within/under a directory file.
- Do not contain other files.
- In long-format output of ls -l, this type of file is specified by the “-” symbol.

2. Directories – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components. (1) The Filename (2) A unique identification number for the file or directory (called the inode number)

- Branching points in the hierarchical tree.
- Used to organize groups of files.
- May contain ordinary files, special files or other directories.
- Never contain “real” information which you would work with (such as text). Basically, just used for organizing files.

- All files are descendants of the root directory, (named /) located at the top of the tree.

In long-format output of ls -l , this type of file is specified by the “d” symbol. **3. Special Files** – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations. **Device or special files** are used for device Input/Output(I/O) on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory. On UNIX systems there are two flavors of special files for each device, character special files and block special files :

- When a character special file is used for device Input/Output(I/O), data is transferred one character at a time. This type of access is called raw device access.
- When a block special file is used for device Input/Output(I/O), data is transferred in large fixed-size blocks. This type of access is called block device access.

For terminal devices, it's one character at a time. For disk devices though, raw access means reading or writing in whole chunks of data – blocks, which are native to your disk.

- In long-format output of ls -l, character special files are marked by the “c” symbol.
- In long-format output of ls -l, block special files are marked by the “b” symbol.

4. Pipes – UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another. A Unix pipe provides a one-way flow of data. The output or result of the first command sequence is used as the input to the second command sequence. To make a pipe, put a vertical bar (|) on the command line between two commands. For example: **who | wc -l** In long-format output of ls -l , named pipes are marked by the “p” symbol.

5. Sockets – A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework. In essence, it is a stream of data, very similar to network stream (and network sockets), but all the transactions are local to the filesystem. In long-format output of ls -l, Unix sockets are marked by “s” symbol.

6. Symbolic Link – Symbolic link is used for referencing some other file of the file system. Symbolic link is also known as Soft link. It contains a text form of the path to the file it references. To an end user, symbolic link will appear to have its own name, but when you try reading or writing data to this file, it will instead reference these operations to the file it points to. If we delete the soft link itself , the data file would still be there. If we delete the source file or move it to a different location, symbolic file will not function properly. In long-format output of ls -l , Symbolic link are marked by the “l” symbol (that's a lower case L).

Privileged instructions are those that can only be executed by the operating system kernel or a privileged process, such as a device driver. These instructions typically perform

operations that require direct access to hardware or other privileged resources, such as setting up memory mappings or accessing I/O devices. Privileged instructions are executed in kernel mode, which provides unrestricted access to the system resources.

Non-privileged instructions are those that can be executed by any process, including user-level processes. These instructions are typically used for performing computations, accessing user-level resources such as files and memory, and managing process control. Non-privileged instructions are executed in user mode, which provides limited access to system resources and ensures that processes cannot interfere with one another.

The role of the operating system in managing privileged instructions are:

- **Access control:** The operating system employs access control mechanisms to limit access to privileged instructions. These mechanisms are restricted to authorized processes or users with elevated privileges. This guarantees that privileged instructions can only be executed by trusted processes and thwarts unauthorized access of malicious programs to system resources.
- **Memory protection:** By deploying memory protection techniques, the operating system restricts processes from accessing any memory location that belongs to it or other processes. This aids in preventing tampering with the operating system or other processes by ensuring that processes cannot do so. Furthermore, it hinders malevolent programs from resulting in system crashes or risking the system's safety.
- **Interrupt handling:** The execution of privileged instructions like system calls or exceptions is handled by the operating system through interrupt handling to ensure safety and accuracy, so when an interrupt occurs during the execution of a process it's necessary to save its state before transferring control to a suitable handler and then restoring it on completion.
- **Virtualization:** Using virtualization techniques allows the operating system to create a simulated environment where processes can execute privileged instructions without having direct access to the underlying hardware thus creating a more secure and isolated execution environment for privileged instructions by limiting process access to authorized hardware resources only.

Here are some key differences between privileged and non-privileged instructions:

1. Access to resources: Privileged instructions have direct access to system resources, while non-privileged instructions have limited access.
2. Execution mode: Privileged instructions are executed in kernel mode, while non-privileged instructions are executed in user mode.
3. Execution permissions: Privileged instructions require special permissions to execute, while non-privileged instructions do not.

4. Purpose: Privileged instructions are typically used for performing low-level system operations, while non-privileged instructions are used for general-purpose computing.
5. Risks: Because privileged instructions have access to system resources, they pose a higher risk of causing system crashes or security vulnerabilities if not used carefully. Non-privileged instructions are less risky in this regard.

In summary, privileged instructions are used by the operating system kernel and privileged processes to perform low-level system operations, while non-privileged instructions are used by user-level processes for general-purpose computing. The distinction between privileged and non-privileged instructions is an important mechanism for ensuring the security and stability of an operating system.

Privileged Instructions possess the following characteristics :

(i) If any attempt is made to execute a Privileged Instruction in User Mode, then it will not be executed and treated as an illegal instruction. The Hardware traps it in the Operating System.

(ii) Before transferring the control to any User Program, it is the responsibility of the Operating System to ensure that the **Timer** is set to interrupt. Thus, if the timer interrupts then the Operating System regains control.

Thus, any instruction which can modify the contents of the Timer is Privileged Instruction

(iii) Privileged Instructions are used by the Operating System in order to achieve correct operation.

(iv) Various examples of Privileged Instructions include:

- I/O instructions and Halt instructions
- Turn off all Interrupts
- Set the Timer
- Context Switching
- Clear the Memory or Remove a process from the Memory
- Modify entries in the Device-status table

Various examples of Non-Privileged Instructions include:

- Reading the status of Processor
- Reading the System Time
- Generate any Trap Instruction

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used:

1. **Absolute Path name** – In this method, each file is given an **absolute path** name consisting of the path from the root directory to the file. As an example, the path **/usr/ast/mailbox** means that the root directory contains a subdirectory **usr**, which in turn contains a subdirectory **ast**, which contains the file **mailbox**.

Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by '/'. In Windows, the separator is '\'. **Windows** \usr\ast\mailbox **UNIX** /usr/ast/mailbox

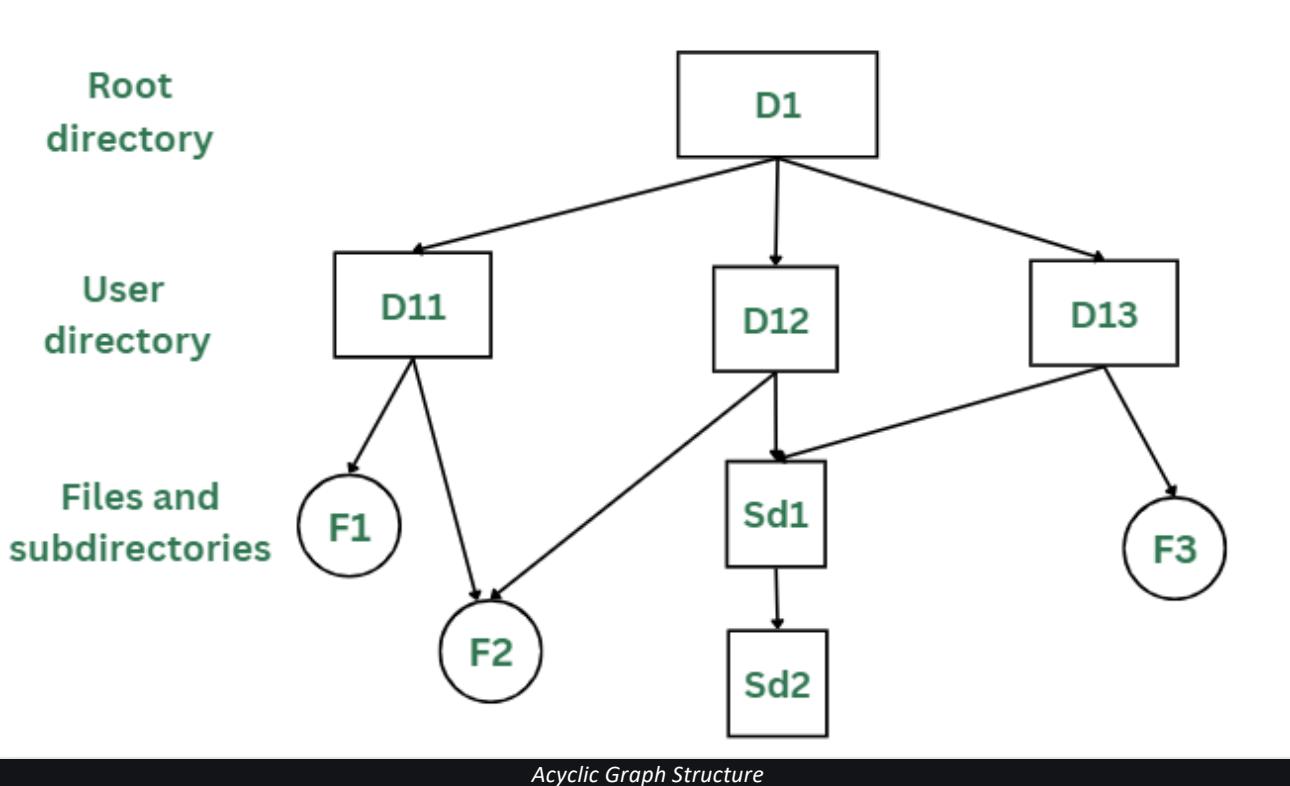
2. **Relative Path name** – This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For **example**, if the current working directory is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox. In other words, the UNIX command : **cp /usr/ast/mailbox /usr/ast/mailbox.bak** and the command : **cp mailbox mailbox.bak** do exactly the same thing if the working directory is /usr/ast.

When to use which approach? Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read /usr/lib/dictionary to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is. Of course, if the spelling checker needs a large number of files from /usr/lib, an alternative approach is for it to issue a system call to change its working directory to /usr/lib, and then use just dictionary as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Acyclic Graph Structure:

As we have seen the above three directory structures, where none of them have the capability to access one file from multiple directories. The file or the subdirectory could be accessed through the directory it was present in, but not from the other directory. This problem is solved in acyclic graph directory structure, where a file in one directory can be accessed from multiple directories. In this way, the files could be shared in between the users. It is designed in a way that multiple directories point to a particular directory or file with the help of links.

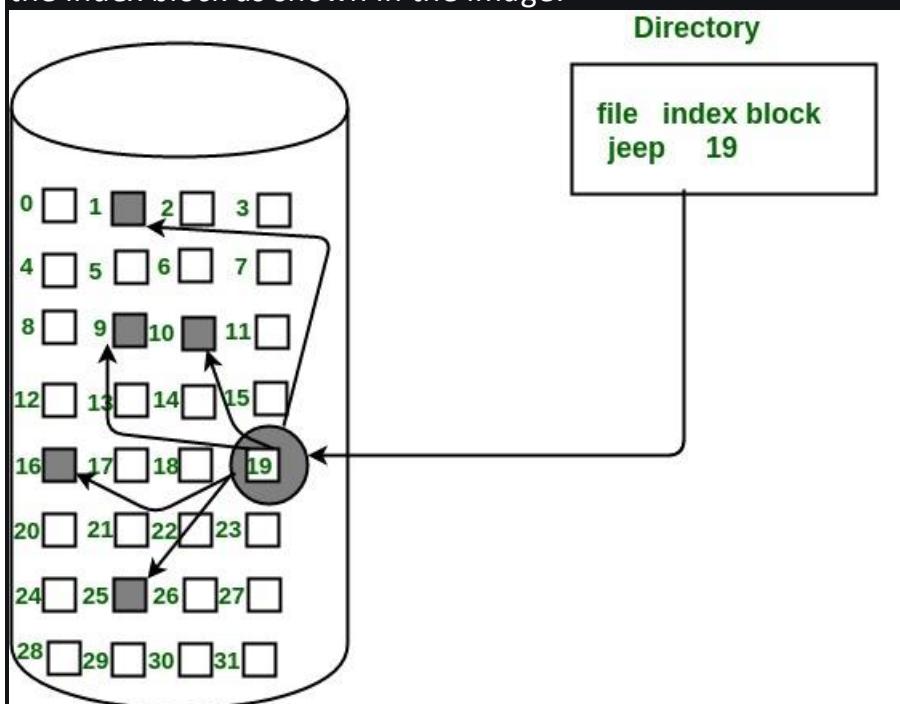
In the below figure, this explanation can be nicely observed, where a file is shared between multiple users. If any user makes a change, it would be reflected to both the users.



Acyclic Graph Structure

Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The i th entry in the index block contains the disk address of the i th file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

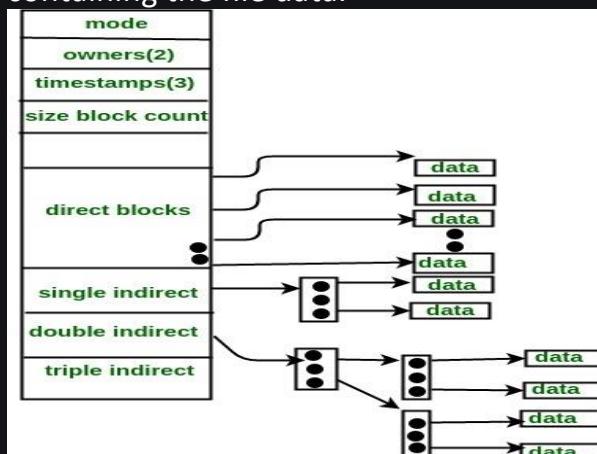
Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

For files that are very large, single index block may not be able to hold all the pointers.

Following mechanisms can be used to resolve this:

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which inturn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3. **Combined Scheme:** In this scheme, a special block called the **Inode (information Node)** contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file *as shown in the image below*. The first few of these pointers in Inode point to the **direct blocks** i.e the pointers contain the addresses of the disk blocks that contain data of the file. The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect. **Single Indirect block** is the disk block that does not contain the file data but the disk address of the blocks that contain the file data. Similarly, **double indirect blocks** do not contain the file data but the disk address of the blocks that contain the address of the blocks containing the file data.



Sequential Access –

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editor and compiler usually access the file in this fashion.

Read and write make up the bulk of the operation on a file. A read operation -*read next*- read the next position of the file and automatically advance a file pointer, which keeps track I/O location. Similarly, for the -*write next*- append to the end of the file and advance to the newly written material.

2.Direct Access –

Another method is *direct access method* also known as *relative access method*. A fixed-length logical record that allows the program to read and write record rapidly. in no particular order. The direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59, and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file.

A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on. **3.Index sequential method –**

It is the other method of accessing a file that is built on the top of the sequential access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index, and then by the help of pointer we access the file directly. **4.Relative Record Access –**

Relative record access is a file access method used in operating systems where records are accessed relative to the current position of the file pointer. In this method, records are located based on their position relative to the current record, rather than by a specific address or key value.

5.Content Addressable Access-

Content-addressable access (CAA) is a file access method used in operating systems that allows records or blocks to be accessed based on their content rather than their address. In this method, a hash function is used to calculate a unique key for each record or block, and the system can access any record or block by specifying its key.

In a computer, memory refers to the physical devices that are used to store programs or data on a temporary or permanent basis. It is a group of registers. Memory are of two types (i) primary memory, (ii) secondary memory. Primary memory is made up of semiconductors, It is also divided into two types, Read-Only Memory (ROM) and Random Access Memory (RAM). Secondary memory is a physical device for the permanent storage of programs and data(Hard disk, Compact disc, Flash drive, etc.).

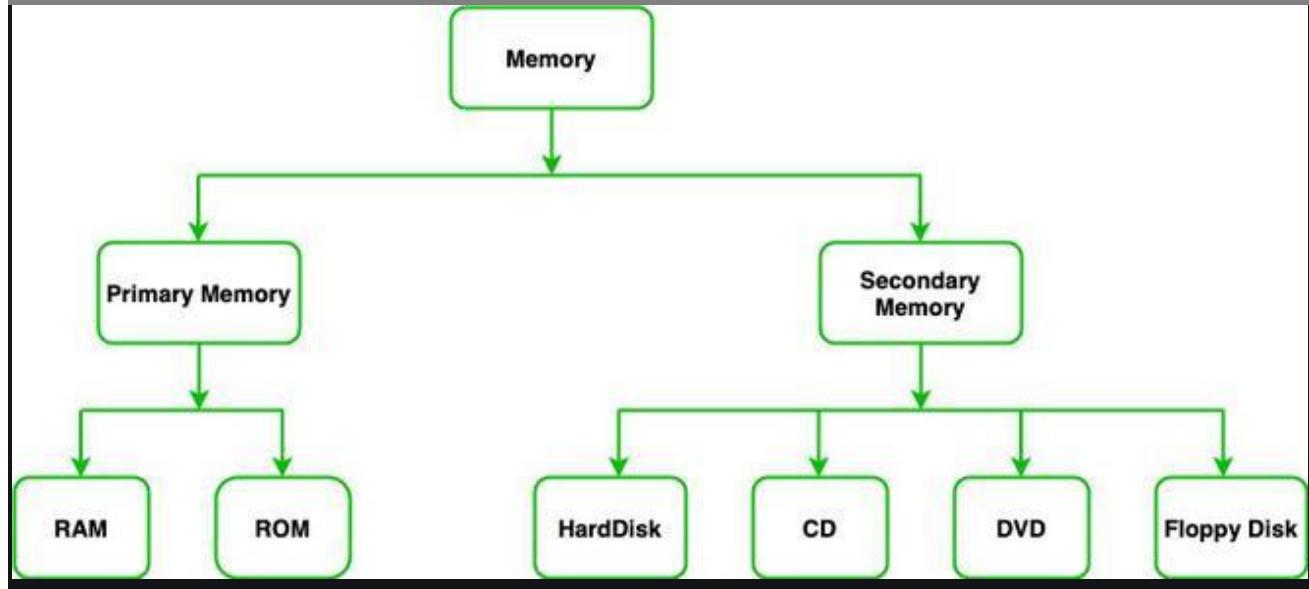
Secondary memory is a type of computer memory that is used to store data and programs that can be accessed or retrieved even after the computer is turned off. Unlike primary memory, which is volatile and temporary, secondary memory is non-volatile and can store data and programs for extended periods of time.

1. Some examples of secondary memory include hard disk drives (HDDs), solid-state drives (SSDs), optical discs (such as CDs and DVDs), and flash memory (such as USB drives and memory cards). These storage devices provide a much larger capacity than primary memory and are typically used to store large amounts of data, such as operating systems, application programs, media files, and other types of digital content.
2. Secondary memory can be classified into two types: magnetic storage and solid-state storage. Magnetic storage devices, such as hard disk drives and magnetic tapes, use magnetic fields to store and retrieve data. Solid-state storage devices, such as solid-state drives and flash memory, use semiconductor-based memory chips to store data.
3. One of the main advantages of secondary memory is its non-volatile nature, which means that data and programs stored on secondary memory can be accessed even after the computer is turned off. Additionally, secondary memory devices provide a large storage capacity, making it possible to store large amounts of data and programs.

However, there are also some disadvantages to secondary memory, such as slower access times and lower read/write speeds compared to primary memory. Additionally, secondary memory devices are often more prone to mechanical failures and data corruption, which can result in data loss.

Overall, secondary memory plays an important role in modern computing systems and is essential for storing large amounts of data and programs.

Learn more



Primary Memory

Primary memory is made up of semiconductors and it is the main memory of the computer system. It is generally used to store data or information on which the computer is currently working, so we can say that it is used to store data temporarily. Data or information is lost when the systems are off. It is also divided into two types:

(i). Read-Only Memory (ROM)

(ii). Random Access Memory (RAM).

1. Random Access Memory: Primary memory is also called internal memory. This is the main area in a computer where data, instructions, and information are stored. Any storage location in this memory can be directly accessed by the Central Processing Unit. As the CPU can randomly access any storage location in this memory, it is also called Random Access Memory or RAM. The CPU can access data from RAM as long as the computer is switched on. As soon as the power to the computer is switched off, the stored data and instructions disappear from RAM. Such type of memory is known as volatile memory. RAM is also called read/write memory.

2. Read-Only Memory: Read-Only Memory (ROM) is a type of primary memory from which information can only be read. So it is also known as Read-Only Memory. ROM can be directly accessed by the Central Processing Unit. But, the data and instructions stored in ROM are retained even when the computer is switched off OR we can say it holds the data after being switched off. Such type of memory is known as non-volatile memory.

Secondary Memory

We have read so far, that primary memory is volatile and has limited capacity. So, it is important to have another form of memory that has a larger storage capacity and from which data and programs are not lost when the computer is turned off. Such a type of memory is called secondary memory. In secondary memory, programs and data are stored. It is also called auxiliary memory. It is different from primary memory as it is not directly accessible through the CPU and is non-volatile. Secondary or external storage devices have a much larger storage capacity and the cost of secondary memory is less as compared to primary memory.

Use of Secondary memory

Secondary memory is used for different purposes but the main purposes of using secondary memory are:

- **Permanent storage:** As we know that primary memory stores data only when the power supply is on, it loses data when the power is off. So we need a secondary memory to store data permanently even if the power supply is off.
- **Large Storage:** Secondary memory provides large storage space so that we can store large data like videos, images, audios, files, etc permanently.
- **Portable:** Some secondary devices are removable. So, we can easily store or transfer data from one computer or device to another.

Types of Secondary memory

1. Fixed storage

In secondary memory, a fixed storage is an internal media device that is used to store data in a computer system. Fixed storage is generally known as fixed disk drives or hard drives. Generally, the data of the computer system is stored in a built-in fixed storage device. Fixed storage does not mean that you can not remove them from the computer system, you can remove the fixed storage device for repairing, for the upgrade, or for maintenance, etc. with the help of an expert or engineer.

Types of fixed storage:

Following are the types of fixed storage:

- Internal flash memory (rare)
- SSD (solid-state disk)
- Hard disk drives (HDD)

2. Removable storage

In secondary memory, removable storage is an external media device that is used to store data in a computer system. Removable storage is generally known as disks drives or external drives. It is a storage device that can be inserted or removed from the computer according to our requirements. We can easily remove them from the computer system while the computer system is running. Removable storage devices are portable so we can easily transfer data from one computer to another. Also, removable storage devices provide the fast data transfer rates associated with storage area networks (SANs).

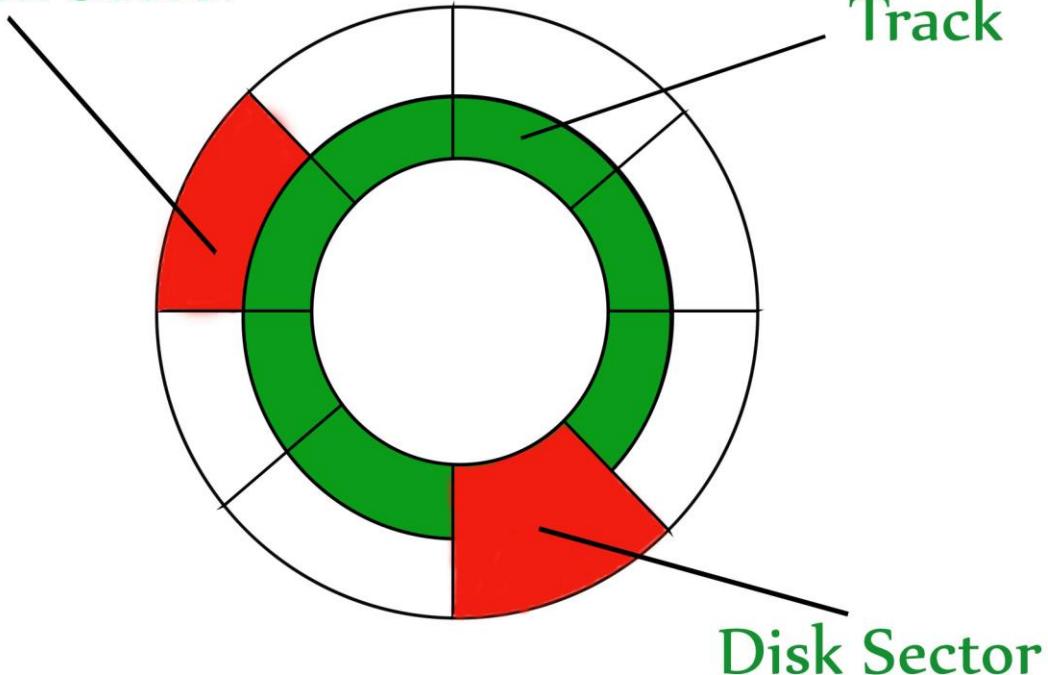
Types of Removable Storage:

- Optical discs (like CDs, DVDs, Blu-ray discs, etc.)
- Memory cards
- Floppy disks
- Magnetic tapes
- Disk packs
- Paper storage (like punched tapes, punched cards, etc.)

4.3 What is a Hard Disk Drive?

A hard disk is a memory storage device that looks like this:

Track Sector



The disk is divided into tracks. Each track is further divided into sectors. The point to be noted here is that outer tracks are bigger in size than the inner tracks but they contain the same number of sectors and have equal storage capacity. This is because the storage density is high in sectors of the inner tracks whereas the bits are sparsely arranged in sectors of the outer tracks. Some space of every sector is used for formatting. So, the actual capacity of a sector is less than the given capacity.

Read-Write(R-W) head moves over the rotating [hard disk](#). It is this Read-Write head that performs all the read and writes operations on the disk and hence, the position of the R-W head is a major concern. To perform a read or write operation on a memory location, we need to place the R-W head over that position. Some important terms must be noted here:

1. **Seek time** – The time taken by the R-W head to reach the desired track from its current position.
2. **Rotational latency** – Time is taken by the sector to come under the R-W head.
3. **Data transfer time** – Time is taken to transfer the required amount of data. It depends upon the rotational speed.
4. **Controller time** – The processing time taken by the controller.
5. **Average Access time** – seek time + Average Rotational latency + data transfer time + controller time.

Note: Average Rotational latency is mostly $1/2 * (\text{Rotational latency})$.

In questions, if the seek time and controller time are not mentioned, take them to be zero.

If the amount of data to be transferred is not given, assume that no data is being transferred. Otherwise, calculate the time taken to transfer the given amount of data.

The average rotational latency is taken when the current position of the R-W head is not given. Because the R-W may be already present at the desired position or it might take a whole rotation to get the desired sector under the R-W head. But, if the current position of the R-W head is given then the rotational latency must be calculated.

4.4 How do Hard Disk Drives Work?

Most basic hard drives are made up of numerous [disk platters](#), which are circular disks composed of aluminum, glass, or ceramic that are arranged around a spindle inside a sealed chamber. The platter is spun by a motor attached to the spindle. The chamber also contains the read/write heads, which use a magnetic head to record information to and from tracks on the platters. The disks are additionally covered in a thin magnetic coating. The platters rotate at up to 15,000 rotations per minute by the motor. A second motor regulates the location of the read and write heads that magnetically record and read information on each platter as the platters rotate. The compartments that store the data can be spread out all over the hard disk.

4.5 Hard Disk Drive Storage Capacity

- Three storage options 16 GB, 32 GB, and 64 GB. This is one of the lowest [HDD](#) storage space ranges and is often seen in older and smaller devices.
- There are two sizes 120 GB and 256 GB. This category is commonly regarded as an entry level for HDD devices like as [laptops or desktops](#).
- 500 GB, 1 TB, and 2 TB are available. HDD storage of 500 GB or more is often deemed adequate for the average user. With this much capacity, users can most certainly save all of their music, photos, videos, and other information. Individuals with large-file games should find 1 TB to 2 TB of HDD capacity adequate.
- More than 2 TB of storage. Anything with more than 2 TB of HDD space is appropriate for users who work with high-resolution files.
- Recently, the highest capacity of HDD is 20 TB.

4.6 History of HDD

The hard disk was developed in 1953 by IBM developers who were looking for a solution to offer low-cost random access to large amounts of data. The first disk drives, which had a capacity of 3.75 MB and were the size of [refrigerators](#), started to be shipped in 1956. Another early supplier of hard disk drive technology was Memorex, followed by Seagate Technology and Western Digital.

The size of hard disk drives has been getting smaller as technology has advanced. The 2.5-inch and 3.5-inch form factors were introduced and standardized in personal computers by the middle of the 1980s.

The storage capacity of latest hard disk drives is in the terabyte range, compared to the initial hard disk drives' capacity of megabytes. HGST, formerly known as Hitachi Global Storage Technologies, is a Western Digital brand released the first 1 TB hard drives in 2007. In 2015, HGST introduced the first 10 TB hard drive. And in 2021, Western Digital introduced two 20 TB HDDs.

4.7 Features of HDD

- **Non-volatile:** HDD is a non-volatile memory device, which means that the data stored on it persists even when the power is turned off. This makes it an ideal storage medium for long-term data storage.
- **High Capacity:** HDDs can store a large amount of data. Modern HDDs can store terabytes of data, making them an ideal choice for storing large files such as videos, photos, and audio recordings.
- **Relatively Slow Speed:** Compared to primary memory devices such as [RAM](#), HDDs are relatively slow. The data access time for an HDD is typically measured in milliseconds, while the access time for RAM is measured in nanoseconds. This makes HDDs better suited for long-term storage rather than for frequently accessed data.
- **Mechanical Parts:** Unlike [solid-state drives \(SSDs\)](#), HDDs have mechanical parts that can wear out over time, leading to reduced performance or even failure. HDDs contain spinning disks and moving read/write heads, which can be susceptible to damage if the drive is bumped or dropped.
- **Cost-effective:** HDDs are generally less expensive than SSDs, making them a popular choice for budget-conscious users. This is particularly true for larger capacity drives.
- **Reliable:** While HDDs are not as reliable as SSDs due to their mechanical components, they are still considered to be a reliable storage medium. HDDs are designed to withstand heavy use and are often used in enterprise-level storage solutions.

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly as:

1. **Bitmap or Bit vector** – A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: *0 indicates that the block is allocated* and 1 indicates a free block. The given instance of disk blocks on the disk in *Figure 1* (where green blocks are allocated)

can be represented by a bitmap of 16 bits as: **0000111000000110**.

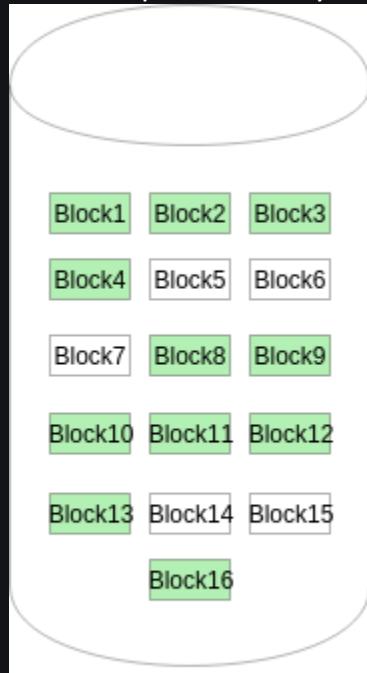


Figure - 1

Advantages –

- Simple to understand.
- Finding the first free block is efficient. It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.

2. **Linked List** – In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in

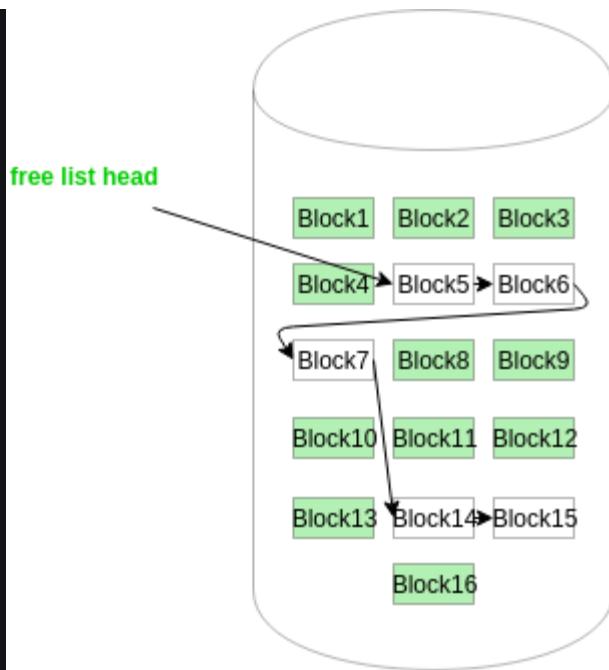


Figure - 2

memory.

In *Figure-2*, the free space

list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list. A drawback of this method is the I/O required for free space list traversal.

3. **Grouping** – This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first $n-1$ blocks are actually free and the last block contains the address of next free n blocks. An **advantage** of this approach is that the addresses of a group of free disk blocks can be found easily.
4. **Counting** – This approach stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block. Every entry in the list would contain:

1. Address of first free disk block
2. A number n

SPOOL is an acronym for **simultaneous peripheral operations on-line**. It is a kind of buffering mechanism or a process in which data is temporarily held to be used and executed by a device, program or the system. Data is sent to and stored in memory or other volatile storage until the program or computer requests it for execution.

4.7.1.1 Need of Spooling

Peripheral equipment such as printers and punch card readers are often slow relative to the performance of the rest of the system, creating a bottleneck in the I/O process. This is where spooling comes in. Spooling resolves this by accumulating data, instructions, and processes from multiple sources in a request queue, which is then processed in a first-in, first-out (FIFO) manner.

4.7.1.2 How Does Spooling Work:

Spooling can be implemented using the computer's physical memory, buffers, or device-specific interrupts. The most common applications of spooling are found in I/O devices like keyboards, printers, and mice. For example, when a document is sent to a printer, it is first stored in the printer's spooler or memory. When the printer is ready, it fetches the data from the spool and prints it.

Batch processing systems also use spooling to maintain a queue of ready-to-run jobs, which can be started as soon as the system has the resources to process them.

Additionally, spooling is capable of overlapping I/O operation for one job with processor operations for another job, allowing multiple processes to write documents to a print queue without waiting and resume with their work.

Applications/Implementations of Spool:

1) The most common can be found in I/O devices like keyboard printers and mouse. For example, In printer, the documents/files that are sent to the printer are first stored in the memory or the printer spooler. Once the printer is ready, it fetches the data from the spool and prints it.

Ever experienced a situation when suddenly for some seconds your mouse or keyboard stops working? Meanwhile, we usually click again and again here and there on the screen to check if its working or not. When it actually starts working, what and wherever we pressed during its hang state gets executed very fast because all the instructions got stored in the respective device's spool.

2) A batch processing system uses spooling to maintain a queue of ready-to-run jobs which can be started as soon as the system has the resources to process them.

3) Spooling is capable of overlapping I/O operation for one job with processor operations for another job. i.e. multiple processes can write documents to a print queue without waiting and resume with their work.

4) E-mail: an email is delivered by a MTA (Mail Transfer Agent) to a temporary storage area where it waits to be picked up by the MA (Mail User Agent)

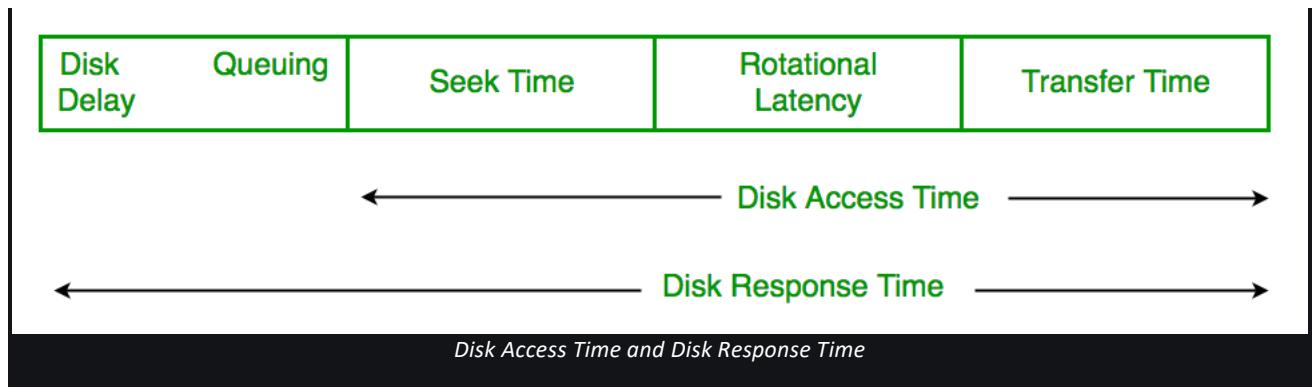
5) Can also be used for generating Banner pages (these are the pages used in computerized printing in order to separate documents from each other and to identify e.g. the originator of the print request by username, an account number or a bin for pickup. Such pages are used in office environments where many people share the small number of available resources).

SPOOLING	BUFFERING	
Basic Difference	It overlaps the input/output of one job with the execution of another job.	It overlaps the input/output of one job with the execution of the same job.
Full form (stands for)	Simultaneous peripheral operation online	No full form
Efficiency	Spooling is more efficient than buffering.	Buffering is less efficient than spooling.
Consider Size	It considers disk as a huge spool or buffer.	Buffer is a limited area in main memory.
Remote Processing	It can process data at remote places.	It does not support remote processing.
Implementation	Implemented using spoolers which manage input/output requests and allocate resources as needed	Implemented through software or hardware-based mechanisms such as circular buffers or FIFO queues
Capacity	Can handle large amounts of data since spooled data is stored on disk or other external storage	Limited by the size of memory available for buffering
Error handling	Since data is stored on external storage, spooling can help recover from system crashes or other errors	Errors can occur if buffer overflow happens, which can cause data loss or corruption
Complexity	More complex than buffering since spooling requires additional software to manage input/output requests	Less complex than spooling since buffering is a simpler technique for managing data transfer

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more requests may be far from each other so this can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

4.8 Key Terms Associated with Disk Scheduling

- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or written. So the disk scheduling algorithm that gives a minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of the disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and the number of bytes to be transferred.
- **Disk Access Time:**



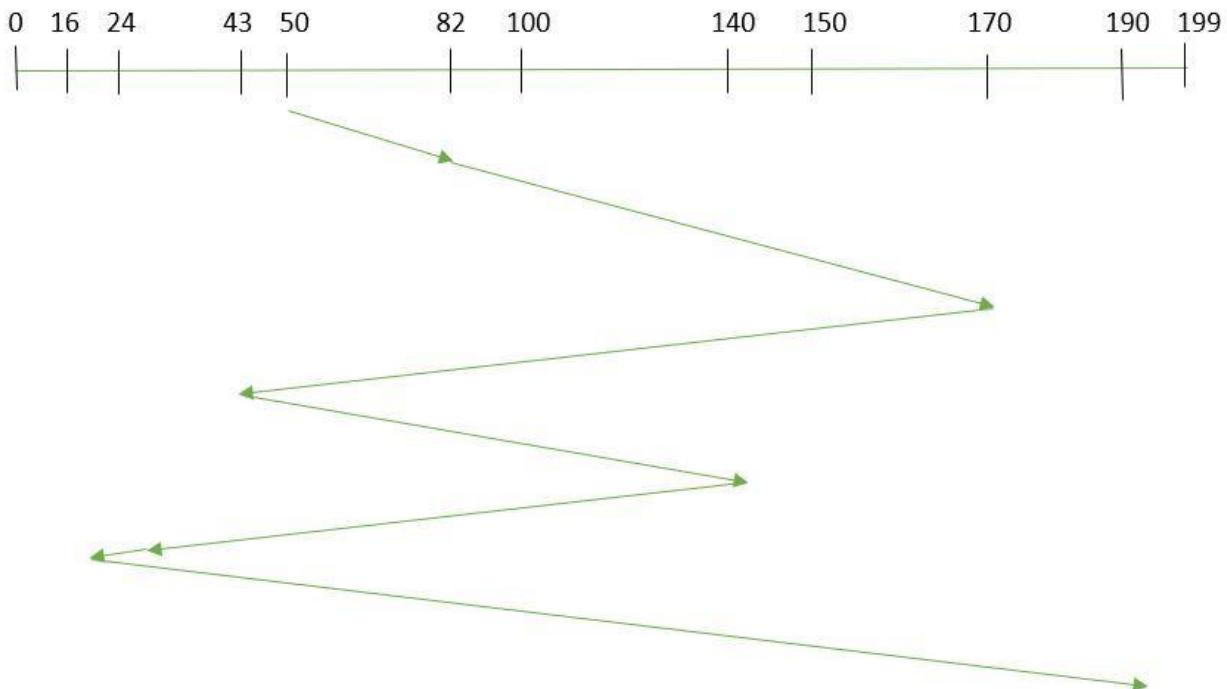
- **Disk Response Time:** Response Time is the average time spent by a request waiting to perform its I/O operation. The average *Response time* is the response time of all requests. *Variance Response Time* is the measure of how individual requests are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

4.9 Disk Scheduling Algorithms

There are several Disk Several Algorithms. We will discuss each one of them.

FCFS (First Come First Serve)

FCFS is the simplest of all Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.



Suppose the order of request is- (82,170,43,140,24,16,190)
And current position of Read/Write head is: 50

Advantages of FCFS

Here are some of the advantages of First Come First Serve.

- Every request gets a fair chance
- No indefinite postponement

Disadvantages of FCFS

Here are some of the disadvantages of First Come First Serve.

- Does not try to optimize seek time

- May not provide the best possible service

SSTF (Shortest Seek Time First)

In [SSTF \(Shortest Seek Time First\)](#), requests having the shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of the system. Let us understand this with the help of an example.

So,

Advantages of Shortest Seek Time First

Here are some of the advantages of Shortest Seek Time First.

- The average Response Time decreases
- Throughput increases

Disadvantages of Shortest Seek Time First

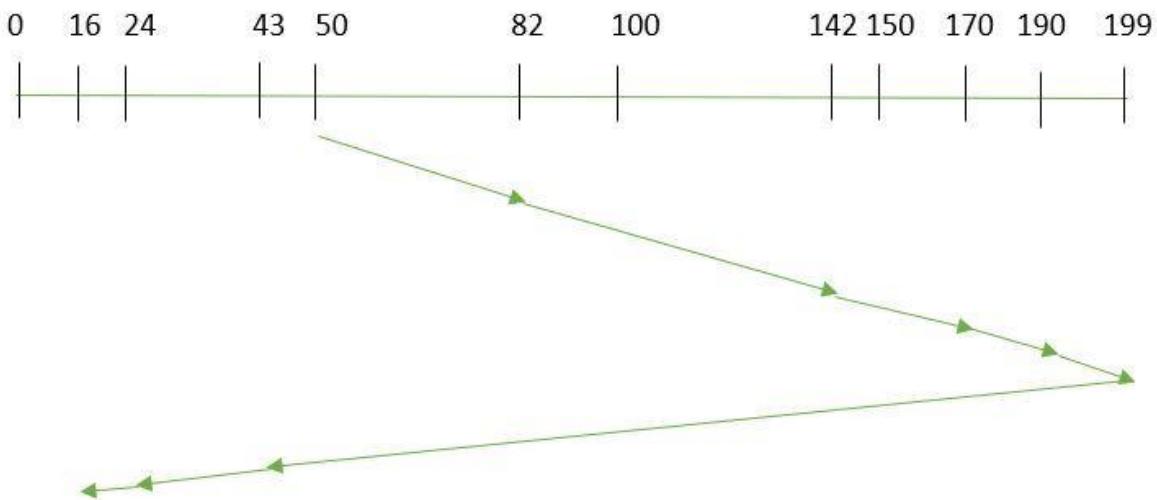
Here are some of the disadvantages of Shortest Seek Time First.

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has a higher seek time as compared to incoming requests
- The high variance of response time as SSTF favors only some requests

SCAN

In the [SCAN algorithm](#) the disk arm moves in a particular direction and services the requests coming in its path and after reaching the end of the disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and is hence also known as an **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example:



SCAN Algorithm

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “**towards the larger value**”. Therefore, the total overhead movement (total distance covered by the disk arm) is calculated as

Advantages of SCAN Algorithm

Here are some of the advantages of the SCAN Algorithm.

- High throughput
- Low variance of response time
- Average response time

Disadvantages of SCAN Algorithm

Here are some of the disadvantages of the SCAN Algorithm.

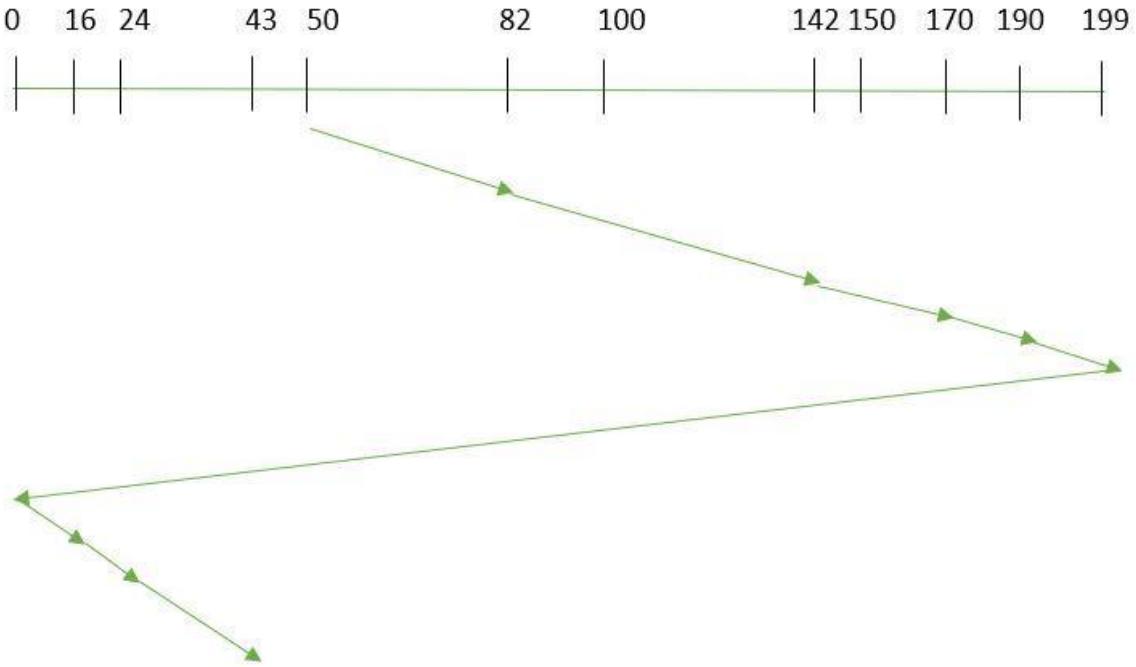
- Long waiting time for requests for locations just visited by disk arm

C-SCAN

In the [SCAN algorithm](#), the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in the CSCAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to the SCAN algorithm hence it is known as C-SCAN (Circular SCAN).

4.9.1.1 Example:



Circular SCAN

Suppose the requests to be addressed are -82, 170, 43, 140, 24, 16, 190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “**towards the larger value**”. So, the total overhead movement (total distance covered by the disk arm) is calculated as:

Advantages of C-SCAN Algorithm

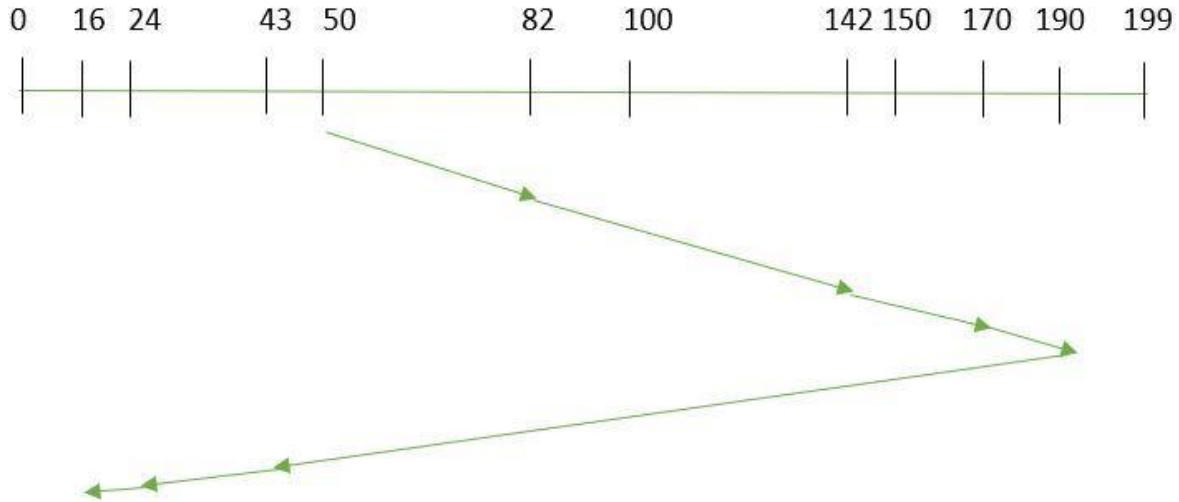
Here are some of the advantages of C-SCAN.

- Provides more uniform wait time compared to SCAN.

LOOK

[LOOK Algorithm](#) is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

4.9.1.2 Example:



LOOK Algorithm

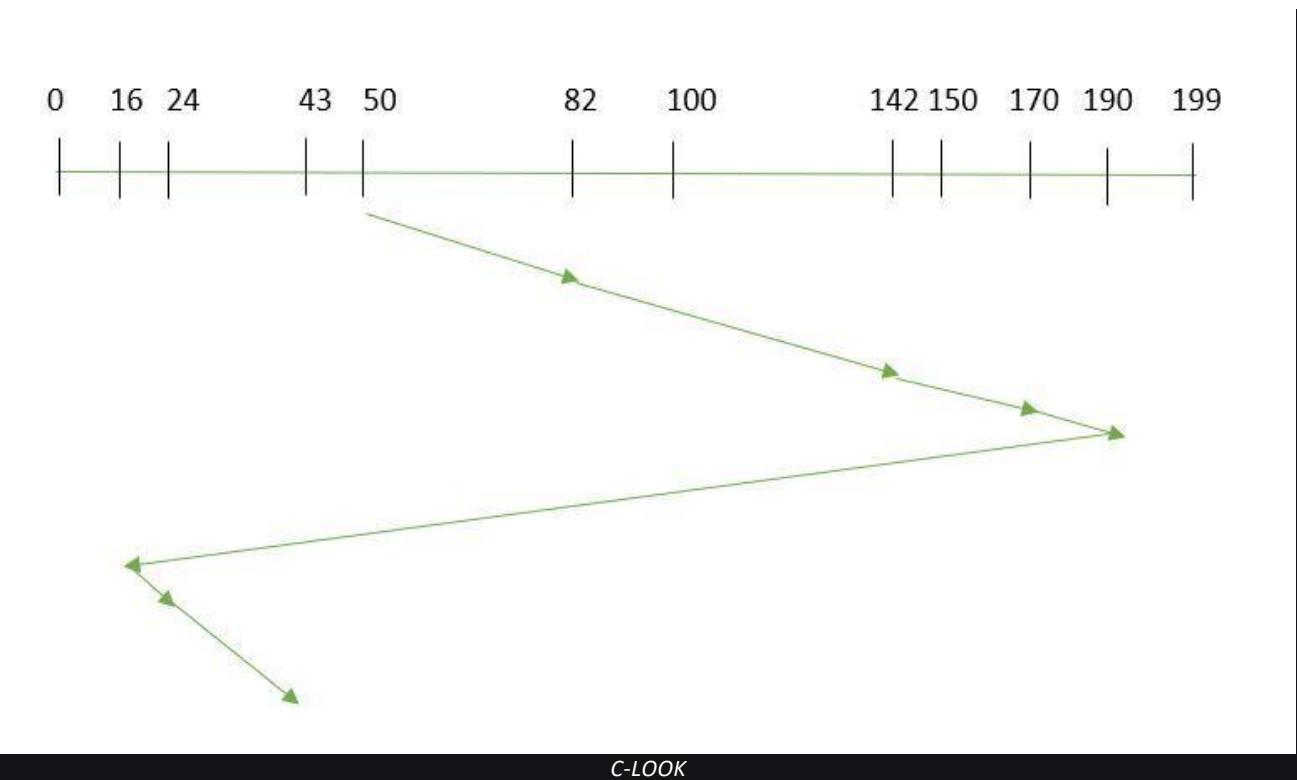
Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “**towards the larger value**”. So, the total overhead movement (total distance covered by the disk arm) is calculated as:

C-LOOK

As LOOK is similar to the SCAN algorithm, in a similar way, C-LOOK is similar to the CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end’s last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

4.9.1.3 Example:

1. Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “**towards the larger value**”



So, the total overhead movement (total distance covered by the disk arm) is calculated as

RSS (Random Scheduling)

It stands for Random Scheduling and just like its name it is natural. It is used in situations where scheduling involves random attributes such as random processing time, random due dates, random weights, and stochastic machine breakdowns this algorithm sits perfectly. Which is why it is usually used for analysis and simulation.

LIFO (Last-In First-Out)

In LIFO (Last In, First Out) algorithm, the newest jobs are serviced before the existing ones i.e. in order of requests that get serviced the job that is newest or last entered is serviced first, and then the rest in the same order.

Advantages of LIFO (Last-In First-Out)

Here are some of the advantages of the Last In First Out Algorithm.

- Maximizes locality and resource utilization
- Can seem a little unfair to other requests and if new requests keep coming in, it cause starvation to the old and existing ones.

N-STEP SCAN

It is also known as the [N-STEP LOOK](#) algorithm. In this, a buffer is created for N requests. All requests belonging to a buffer will be serviced in one go. Also once the buffer is full no new requests are kept in this buffer and are sent to another one. Now, when these N requests are serviced, the time comes for another top N request and this way all get requests to get a guaranteed service

Advantages of N-STEP SCAN

Here are some of the advantages of the N-Step Algorithm.

- It eliminates the starvation of requests completely

F-SCAN

This algorithm uses two sub-queues. During the scan, all requests in the first queue are serviced and the new incoming requests are added to the second queue. All new requests are kept on halt until the existing requests in the first queue are serviced.

Advantages of F-SCAN

Here are some of the advantages of the [F-SCAN](#) Algorithm.

- F-SCAN along with N-Step-SCAN prevents “arm stickiness” (phenomena in I/O scheduling where the scheduling algorithm continues to service requests at or near the current sector and thus prevents any seeking)

Each algorithm is unique in its own way. Overall Performance depends on the number and type of requests.

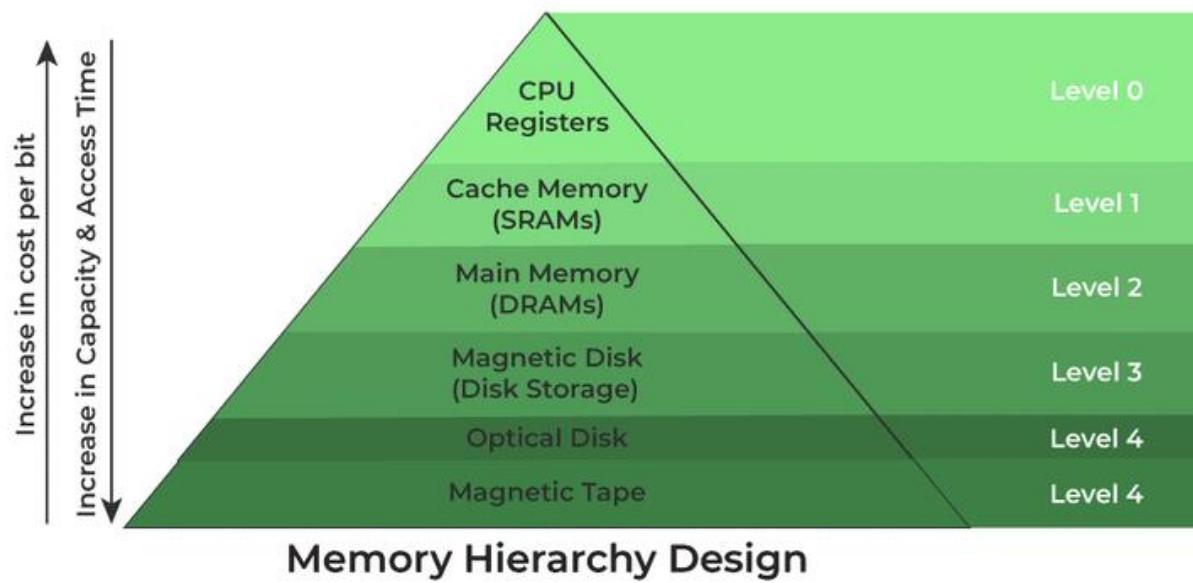
5. Memory Management

Memory Hierarchy is one of the most required things in [Computer Memory](#) as it helps in optimizing the memory available in the computer. There are multiple levels present in the memory, each one having a different size, different cost, etc. Some types of memory like cache, and main memory are faster as compared to other types of memory but they are having a little less size and are also costly whereas some memory has a little higher storage value, but they are a little slower. Accessing of data is not similar in all types of memory, some have faster access whereas some have slower access.

5.1 Types of Memory Hierarchy

This Memory Hierarchy Design is divided into 2 main types:

- **External Memory or Secondary Memory:** Comprising of Magnetic Disk, Optical Disk, and Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via an I/O Module.
- **Internal Memory or Primary Memory:** Comprising of Main Memory, Cache Memory & [CPU registers](#). This is directly accessible by the processor.



5.2 Memory Hierarchy Design

1. Registers

[Registers](#) are small, high-speed memory units located in the CPU. They are used to store the most frequently used data and instructions. Registers have the fastest access time and the smallest storage capacity, typically ranging from 16 to 64 bits.

2. Cache Memory

Cache memory is a small, fast memory unit located close to the CPU. It stores frequently used data and instructions that have been recently accessed from the main memory. Cache memory is designed to minimize the time it takes to access data by providing the CPU with quick access to frequently used data.

3. Main Memory

Main memory, also known as RAM (Random Access Memory), is the primary memory of a computer system. It has a larger storage capacity than cache memory, but it is slower. Main memory is used to store data and instructions that are currently in use by the CPU.

5.2.1.1 Types of Main Memory

- **Static RAM:** Static RAM stores the binary information in flip flops and information remains valid until power is supplied. It has a faster access time and is used in implementing cache memory.
- **Dynamic RAM:** It stores the binary information as a charge on the capacitor. It requires refreshing circuitry to maintain the charge on the capacitors after a few milliseconds. It contains more memory cells per unit area as compared to SRAM.

4. Secondary Storage

Secondary storage, such as hard disk drives (HDD) and solid-state drives (SSD), is a non-volatile memory unit that has a larger storage capacity than main memory. It is used to store data and instructions that are not currently in use by the CPU. Secondary storage has the slowest access time and is typically the least expensive type of memory in the memory hierarchy.

5.3 System-Supported Memory Standards

According to the memory Hierarchy, the system-supported memory standards are defined below:

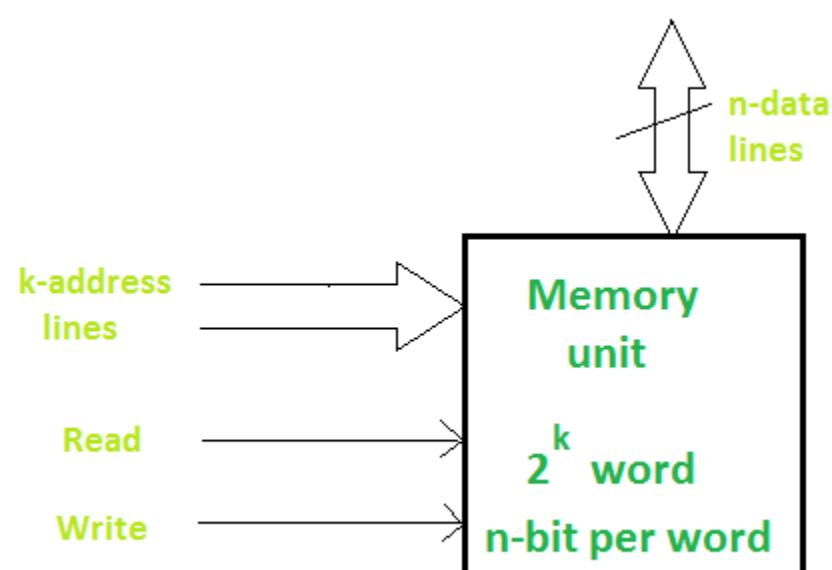
Level	1	2	3	4
Name	Register	Cache	Main Memory	Secondary Memory
Size	<1 KB	less than 16 MB	<16GB	>100 GB
Implementation	Multi-ports	On-chip/SRAM	DRAM (capacitor memory)	Magnetic

Level	1	2	3	4
Access Time	0.25ns to 0.5ns	0.5 to 25ns	80ns to 250ns	50 lakh ns
Bandwidth	20000 to 1 lakh MB	5000 to 15000	1000 to 5000	20 to 150
Managed by	Compiler	Hardware	Operating System	Operating System
Backing Mechanism	From cache	from Main Memory	from Secondary Memory	from ie

A word is a group of bits where a memory unit stores binary information. A word with a group of 8 bits is called a byte.

A memory unit consists of data lines, address selection lines, and control lines that specify the direction of transfer.

The block diagram of a memory unit is shown below:



Learn more

Data lines provide the information to be stored in memory. The control inputs specify the direct transfer. The k-address lines specify the word chosen.

When there are k address lines, 2^k memory words can be accessed.

5.4 Types of Computer Memory

- **Cache memory.** This temporary storage area, known as a cache, is more readily available to the processor than the computer's main memory source. It is also

called *CPU memory* because it is typically integrated directly into the CPU chip or placed on a separate chip with a bus interconnect with the CPU.

- **RAM.** It is one of the parts of the Main memory, also famously known as Read Write Memory. Random Access memory is present on the motherboard and the computer's data is temporarily stored in RAM. As the name says, RAM can help in both Read and write.
- **D RAM (Dynamic RAM):** D RAM uses capacitors and transistors and stores the data as a charge on the capacitors. They contain thousands of memory cells. It needs refreshing of charge on capacitor after a few milliseconds. This memory is slower than S RAM.
- **S RAM (Static RAM):** S RAM uses transistors and the circuits of this memory are capable of retaining their state as long as the power is applied. This memory consists of the number of flip flops with each flip flop storing 1 bit. It has less access time and hence, it is faster.
- **ROM:** ROM full form is Read Only Memory. ROM is a non volatile memory and it is used to store important information which is used to operate the system. We can only read the programs and data stored on it and can not modify or delete it.
- **MROM(Masked ROM):** Hard-wired devices with a pre-programmed collection of data or instructions were the first ROMs. Masked ROMs are a type of low-cost ROM that works in this way.
- **PROM (Programmable Read Only Memory):** This read-only memory is modifiable once by the user. The user purchases a blank PROM and uses a PROM program to put the required contents into the PROM. Its content can't be erased once written.
- **EPROM (Erasable Programmable Read Only Memory):** EPROM is an extension to PROM where you can erase the content of ROM by exposing it to Ultraviolet rays for nearly 40 minutes.
- **EEPROM (Electrically Erasable Programmable Read Only Memory):** Here the written contents can be erased electrically. You can delete and reprogramme EEPROM up to 10,000 times. Erasing and programming take very little time, i.e., nearly 4 -10 ms(milliseconds). Any area in an EEPROM can be wiped and programmed selectively.
- **Virtual memory.** A memory management technique where secondary memory can be used as if it were a part of the main memory. Virtual memory uses hardware and software to enable a computer to compensate for physical memory shortages by temporarily transferring data from RAM to disk storage.

5.5 Units of Memory

Memory units are used to measure the size and represent data. Some of the commonly used memory units are:

1. Bit

The first memory location in a computer is bit. The smallest measurement unit for data held in [primary memory](#) and [storage devices](#) is a bit. Out of the binary values 0 and 1, a bit can only have one.

- The smallest measurement unit for data in primary memory and storage devices.
- Represents binary values 0 and 1.

2. Nibble

- It means the group of 4 bits.

3. Word

It is a fixed number of bits, it is different from computer to computer, but the same for each device. Compute store information in the form of words.

- A fixed number of bits that varies across computers but remains consistent within each device.
- Used to store information in computers.

4. Bytes

The fundamental unit used to measure data is the byte. It has 8 bits in it. A byte can therefore represent 2^8 or 256 values. They determine the size of files, documents, photos, and other kinds of data.

- The fundamental unit for measuring data, consisting of 8 bits.
- Represents 256 values and determines file, document, photo, and data sizes.

5. Kilobyte

1024 bytes is equal to one kilobyte. It is widely used to denote small file sizes and data storage capacities. One kilobyte can hold a small image or around 1024 characters of text. It frequently shows up in text documents, [spreadsheets](#), and small image files.

- Equal to 1024 bytes.
- Denotes small file sizes and storage capacities.
- Can hold small images or around 1024 characters of text.

6. Megabyte

A megabyte is 1024 kilobytes in size. It contains more info as compared to a kilobyte. A megabyte can hold longer texts, high-resolution images, and short audio clips. It is used to calculate the size of files comprising music and short films, [software](#) packages, and documents. Megabytes are still important and frequently used, even though larger units of measurement are being used more frequently as a result of the growing number of data files.

- Comprising 1024 kilobytes.
- Contains more information compared to a kilobyte.
- Holds longer texts, high-resolution images, and short audio clips.
- Measures file sizes of music, short films, software packages, and documents.

7. Gigabyte

1024 megabytes is equal to one gigabyte. It has a substantial amount of data storage space. Larger files, such full photo albums, high-definition movies, and software programs

can fit within a gigabit. The storage capabilities of [hard drives](#), [solid-state drives](#), and other forms of data storage devices are routinely assessed utilizing this technique.

- Equal to 1024 megabytes.
- Offers substantial data storage space.
- Suitable for larger files, such as full photo albums, high-definition movies, and software programs.

8. Terabyte

A terabyte is made up of 1024 gigabytes. It has a substantial amount of data storing capacity. A terabyte can hold a lot of data in large databases, massive media collections, and enterprise-level storage systems. It is frequently used by data centers, [cloud storage](#) services, and external hard drives with large storage capacities. As the demand for large-scale data processing and storage grows, terabytes are becoming more and more important.

- Comprising 1024 gigabytes.
- Provides substantial data storing capacity.
- Holds large databases, media collections, and enterprise-level storage systems.

9. Petabyte

A petabyte is a colossal unit of data storage capacity. A petabyte may hold massive amounts of data, including significant video libraries, sizable databases, and sizable collections of high-resolution pictures. It is often used in data centers, cloud storage, and scientific research that uses a lot of data.

- A colossal unit of data storage capacity.
- Stores massive data quantities, like video libraries and large databases.

10. Exabyte (1024 petabytes)

An exabyte is equal to one EB. It has a substantial amount of data storage space. Exabytes can store vast film archives, massive [data warehouses](#), and global internet traffic. It is extensively used in large-scale scientific simulations, [cloud computing](#) infrastructures, and enterprise-level storage systems.

- Equal to 1024 petabytes.
- Holds vast film archives, data warehouses, and global internet traffic.

11. Zettabyte (1024 exabytes)

A zettabyte. It represents a capacity for data storage that is almost unimaginable. Zettabytes have the capacity to store unfathomably large amounts of data, including worldwide [internet](#) content, long-term archival storage, and in-depth global data analysis.

- Represents an almost unimaginable data storage capacity.
- Stores worldwide internet content, long-term archival data, and extensive global analysis.

12. Yottabyte

1024 zettabytes make up a yottabyte (abbreviated YB). It stands for an incredible amount of data storage. Unimaginable amounts of data, such as the equivalent of storing all of the

material on the internet numerous times or tracking vast amounts, may be stored in yottabytes.

- Comprising 1024 zettabytes.
- Stands for an incredible amount of data storage.
- Can hold vast amounts equivalent to storing internet content numerous times.

5.6 Conversations of units

Name	Equal To	Size (In Bytes)
Bit	1 Bit	1/8
Nibble	4 Bits	1/2 (rare)
Byte	8 Bits	1
Kilobyte	1024 Bytes	1024
Megabyte	1024 Kilobytes	1, 048, 576
Gigabyte	1024 Megabytes	1, 073, 741, 824
Terabyte	1024 Gigabytes	1, 099, 511, 627, 776
Petabyte	1024 Terabytes	1, 125, 899, 906, 842, 624
Exabyte	1024 Petabytes	1, 152, 921, 504, 606, 846, 976
Zettabyte	1024 Exabytes	1, 180, 591, 620, 717, 411, 303, 424
Yottabyte	1024 Zettabytes	1, 208, 925, 819, 614, 629, 174, 706, 176

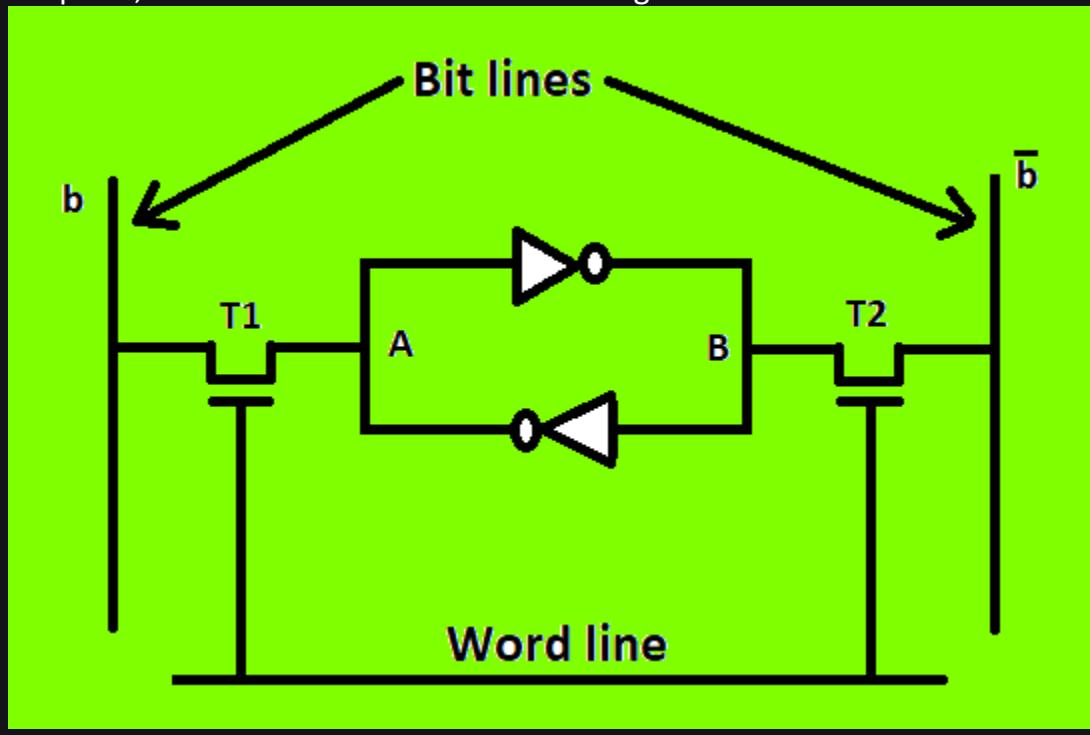
5.7 What is SRAM?

The [SRAM](#) memories consist of circuits capable of retaining the stored information as long as the power is applied. That means this type of memory requires constant power. SRAM memories are used to build [Cache Memory](#).

SRAM Memory Cell

Static memories(SRAM) are memories that consist of circuits capable of retaining their state as long as power is on. Thus this type of memory is called volatile memory. The below figure shows a cell diagram of SRAM. A latch is formed by two inverters connected as shown in

the figure. Two transistors T1 and T2 are used for connecting the latch with two-bit lines. The purpose of these transistors is to act as switches that can be opened or closed under the control of the word line, which is controlled by the address decoder. When the word line is at 0-level, the transistors are turned off and the latch remains its information. SRAM does not require refresh time. For example, the cell is at state 1 if the logic value at point A is 1 and at point, B is 0. This state is retained as long as the word line is not activated.



For the **Read operation**, the word line is activated by the address input to the address decoder. The activated word line closes both the transistors (switches) T1 and T2. Then the bit values at points A and B can transmit to their respective bit lines. The sense/write circuit at the end of the bit lines sends the output to the processor. For the **Write operation**, the address provided to the decoder activates the word line to close both switches. Then the bit value that is to be written into the cell is provided through the sense/write circuit and the signals in bit lines are then stored in the cell.

5.8 What is DRAM?

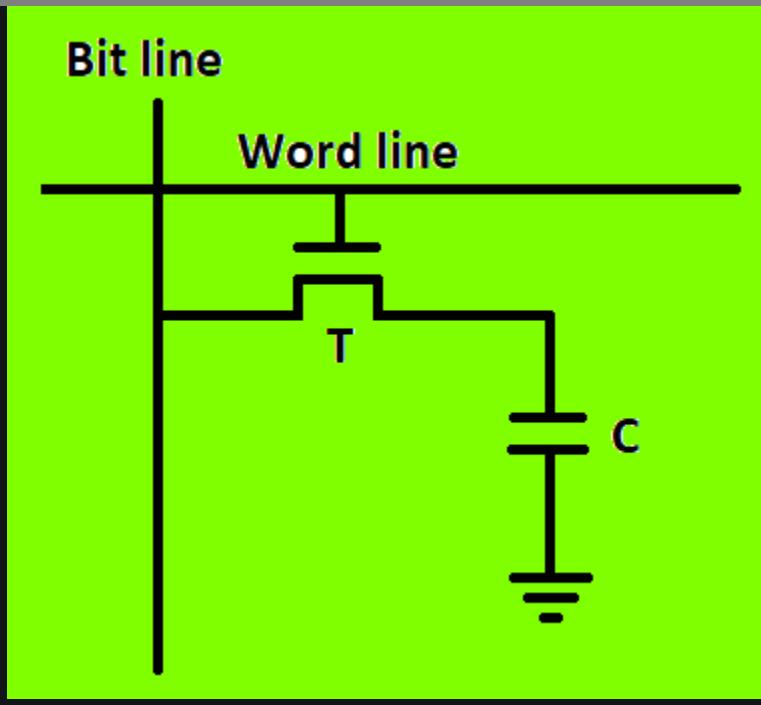
DRAM stores the binary information in the form of electric charges applied to capacitors. The stored information on the capacitors tends to lose over a period of time and thus the capacitors must be periodically recharged to retain their usage. DRAM requires refresh time. The main memory is generally made up of DRAM chips.

DRAM Memory Cell

Though SRAM is very fast, it is expensive because of its every cell requires several transistors. Relatively less expensive RAM is DRAM, due to the use of one transistor and one capacitor in each cell, as shown in the below figure., where C is the capacitor and T is the transistor. Information is stored in a DRAM cell in the form of a charge on a capacitor

and this charge needs to be periodically recharged. For storing information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor. After the transistor is turned off, due to the property of the capacitor, it starts to discharge. Hence, the information stored in the cell can be read correctly only if it is read before the charge on the capacitors drops below some threshold value.

Learn more



5.9 Types of DRAM

There are mainly 5 types of DRAM.

- **Asynchronous DRAM (ADRAM):** The DRAM described above is the asynchronous type of DRAM. The timing of the memory device is controlled asynchronously. A specialized memory controller circuit generates the necessary control signals to control the timing. The CPU must take into account the delay in the response of the memory.
- **Synchronous DRAM (SDRAM):** These RAM chips' access speed is directly synchronized with the CPU's clock. For this, the memory chips remain ready for operation when the CPU expects them to be ready. These memories operate at the CPU-memory bus without imposing wait states. SDRAM is commercially available as modules incorporating multiple SDRAM chips and forming the required capacity for the modules.
- **Double-Data-Rate SDRAM (DDR SDRAM):** This faster version of SDRAM performs its operations on both edges of the clock signal; whereas a standard SDRAM performs its operations on the rising edge of the clock signal. Since they transfer data on both edges of the clock, the data transfer rate is doubled. To

- access the data at a high rate, the memory cells are organized into two groups. Each group is accessed separately.
- **Rambus DRAM (RDRAM):** The RDRAM provides a very high data transfer rate over a narrow CPU-memory bus. It uses various speedup mechanisms, like synchronous memory interface, caching inside the DRAM chips and very fast signal timing. The Rambus data bus width is 8 or 9 bits.
 - **Cache DRAM (CDRAM):** This memory is a special type of DRAM memory with an on-chip cache memory (SRAM) that acts as a high-speed buffer for the main DRAM.

5.10 Difference Between SRAM and DRAM

The below table lists some of the differences between SRAM and DRAM.

SRAM	DRAM
SRAM has a lower access time, so it is faster compared to DRAM.	DRAM has a higher access time, so it is slower than SRAM.
SRAM is costlier than DRAM.	DRAM costs less compared to SRAM.
SRAM requires a constant power supply, which means this type of memory consumes more power.	DRAM offers reduced power consumption due to the fact that the information is stored in the capacitor.
Due to complex internal circuitry, less storage is available compared to the same physical size of a DRAM memory chip.	Due to the small internal circuitry in the one-bit memory cell of DRAM, a large storage capacity is available.
SRAM has low packaging capacity.	DRAM has a high packaging density.
SRAM does not require refresh time.	DRAM requires periodic refresh time.

Prerequisite – [Partition Allocation Methods](#) **Static partition** schemes suffer from the **limitation** of having the fixed number of active processes and the usage of space may also not be optimal. The **buddy system** is a memory allocation and management algorithm that manages memory in **power of two increments**. Assume the memory size is 2^u , suppose a size of S is required.

The Buddy System is a memory allocation technique used in computer operating systems to efficiently allocate and manage memory. The technique is based on dividing the memory into fixed-size blocks, and whenever a process requests memory, the system finds the smallest available block that can accommodate the requested memory size.

Here are the basic steps in the Buddy System memory allocation technique:

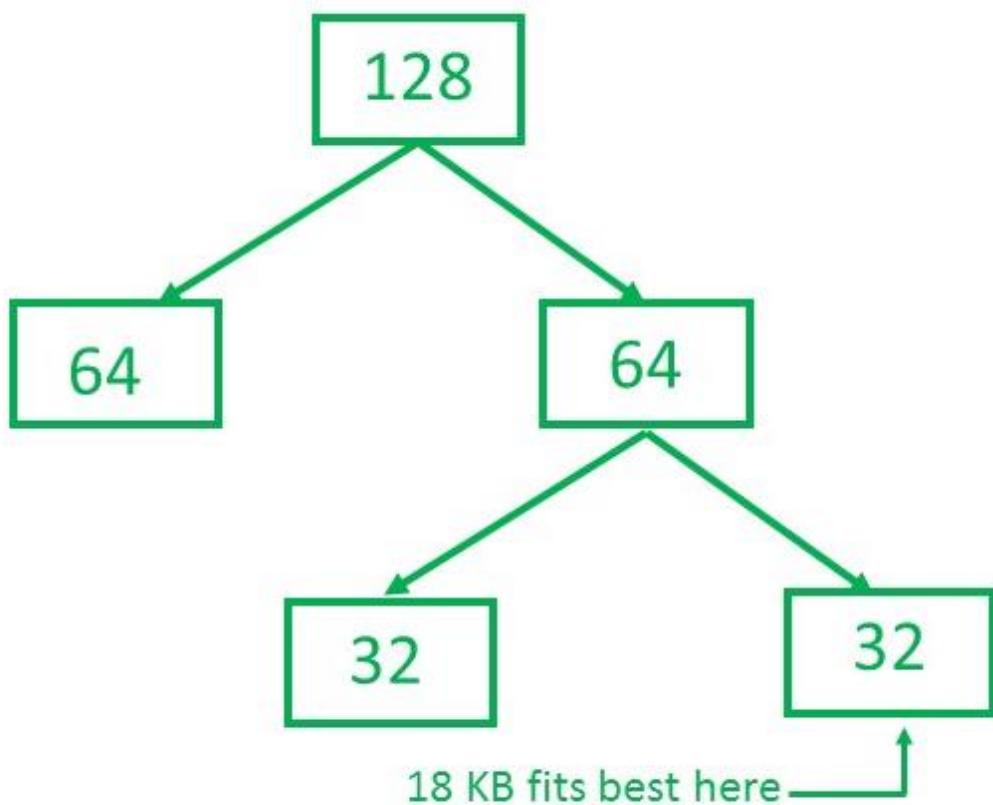
1. The memory is divided into fixed-size blocks that are a power of 2 in size (such as 2, 4, 8, 16, 32, etc. bytes).
2. Each block is labeled with its size and a unique identifier, such as a binary number.
3. Initially, all the memory blocks are free and are linked together in a binary tree structure, with each node representing a block and the tree's leaves representing the smallest available blocks.
4. When a process requests memory, the system finds the smallest available block that can accommodate the requested size. If the block is larger than the requested size, the system splits the block into two equal-sized "buddy" blocks.
5. The system marks one of the buddy blocks as allocated and adds it to the process's memory allocation table, while the other buddy block is returned to the free memory pool and linked back into the binary tree structure.
6. When a process releases memory, the system marks the corresponding block as free and looks for its buddy block. If the buddy block is also free, the system merges the two blocks into a larger block and links it back into the binary tree structure.

The Buddy System technique has several advantages, including efficient use of memory, reduced fragmentation, and fast allocation and deallocation of memory blocks. However, it also has some drawbacks, such as internal fragmentation, where a block may be larger than what the process requires, leading to wastage of memory. Overall, the Buddy System is a useful memory allocation technique in operating systems, particularly for embedded systems with limited memory.

- **If $2^{u-1} < S \leq 2^u$:** Allocate the whole block
- **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block and get out of the loop.

System also keep the record of all the unallocated blocks each and can merge these different size blocks to make one big chunk. **Advantage –**

Example – Consider a system having buddy system with physical address space 128 KB. Calculate the size of partition for 18 KB process. **Solution –**

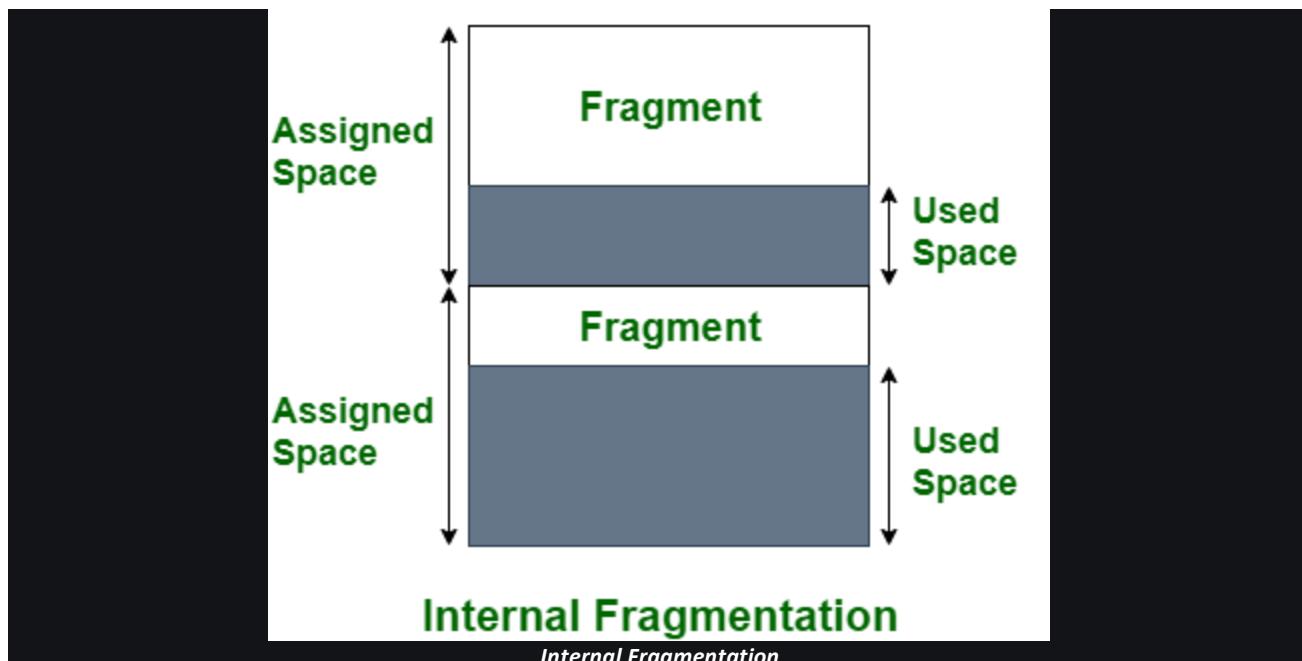


So, size of partition for 18 KB process = 32 KB. It divides by 2, till possible to get minimum block to fit 18 KB.

There are two types of fragmentation in OS which are given as Internal fragmentation and External fragmentation.

1. Internal Fragmentation:

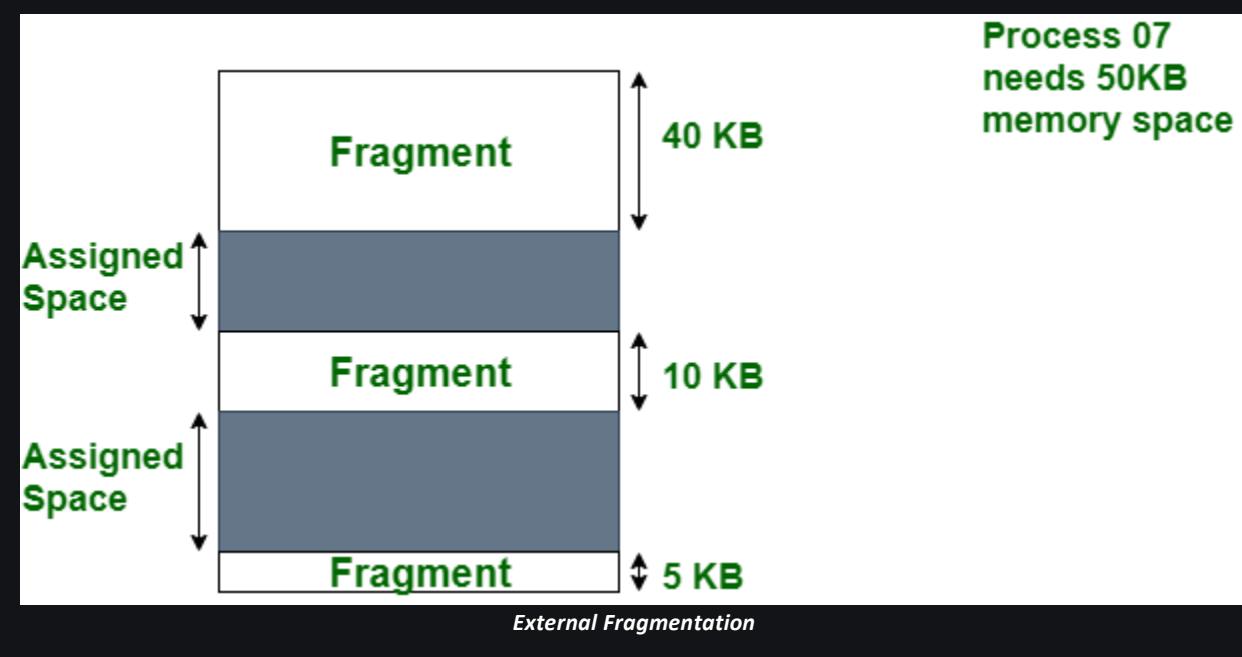
Internal fragmentation happens when the memory is split into mounted-sized blocks. Whenever a method is requested for the memory, the mounted-sized block is allotted to the method. In the case where the memory allotted to the method is somewhat larger than the memory requested, then the difference between 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.



The above diagram clearly shows the internal fragmentation because the difference between memory allocated and required space or memory is called [internal fragmentation](#).

2. 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](#).



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.

Difference between Internal fragmentation and External fragmentation

S.NO	Internal fragmentation	External fragmentation
1.	In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to the method.
2.	Internal fragmentation happens when the method or process is smaller than the memory.	External fragmentation happens when the method or process is removed.
3.	The solution of internal fragmentation is the best-fit block .	The solution to external fragmentation is compaction and paging .
4.	Internal fragmentation occurs when memory is divided into fixed-sized partitions .	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
5.	The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, which is called External fragmentation.
6.	Internal fragmentation occurs with paging and fixed partitioning.	External fragmentation occurs with segmentation and dynamic partitioning .
7.	It occurs on the allocation of a process to a partition greater than the process's requirement. The leftover space causes degradation system performance.	It occurs on the allocation of a process to a partition greater which is exactly the same memory space as it is required.
8.	It occurs in worst fit memory allocation method .	It occurs in best fit and first fit memory allocation method.

In the operating system, the following are four common memory management techniques.

Single contiguous allocation: Simplest allocation method used by MS-DOS. All memory (except some reserved for OS) is available to a process.

Partitioned allocation: Memory is divided into different blocks or partitions. Each process is allocated according to the requirement.

Paged memory management: Memory is divided into fixed-sized units called page frames, used in a virtual memory environment.

Segmented memory management: Memory is divided into different segments (a segment is a logical grouping of the process' data or code). In this management, allocated memory doesn't have to be contiguous.

Most of the operating systems (for example Windows and Linux) use Segmentation with Paging. A process is divided into segments and individual segments have pages.

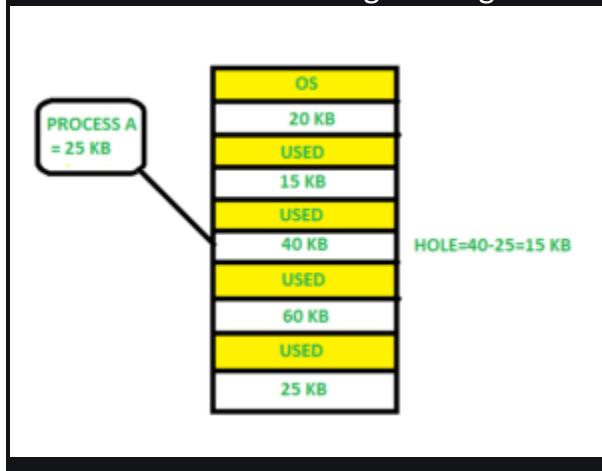
In **Partition Allocation**, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

When it is time to load a process into the main memory and if there is more than one free block of memory of sufficient size then the OS decides which free block to allocate.

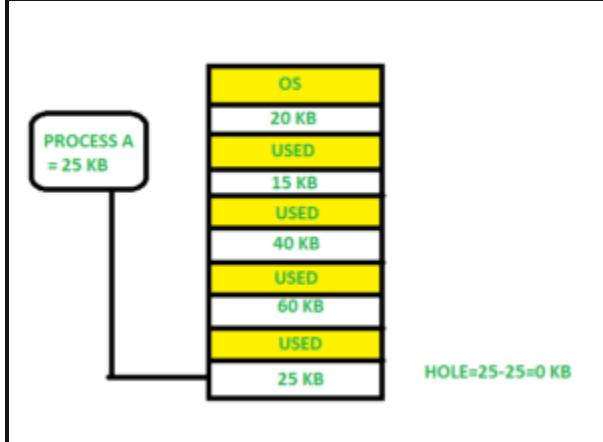
There are different Placement Algorithm:

- A. First Fit
- B. Best Fit
- C. Worst Fit
- D. Next Fit

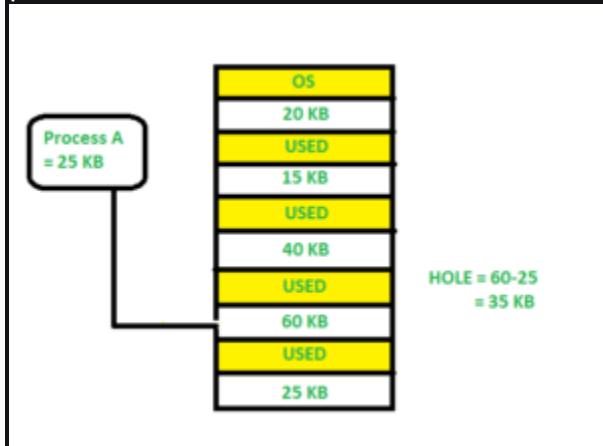
1. First Fit: In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.



2. Best Fit Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. It searches the entire list of holes to find the smallest hole whose size is greater than or equal to the size of the process.



3. Worst Fit Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite to the best-fit algorithm. It searches the entire list of holes to find the largest hole and allocate it to process.



4. Next Fit: Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

Is Best-Fit really best?

Although best fit minimizes the wastage space, it consumes a lot of processor time for searching the block which is close to the required size. Also, Best-fit may perform poorer than other algorithms in some cases. For example, see the below exercise.

Exercise: Consider the requests from processes in given order 300K, 25K, 125K, and 50K. Let there be two blocks of memory available of size 150K followed by a block size 350K. Which of the following partition allocation schemes can satisfy the above requests?

- A) Best fit but not first fit.
- B) First fit but not best fit.

- C) Both First fit & Best fit.
D) neither first fit nor best fit.

Solution: Let us try all options.

Best Fit:

300K is allocated from a block of size 350K. 50 is left in the block.
25K is allocated from the remaining 50K block. 25K is left in the block.
125K is allocated from 150 K block. 25K is left in this block also.
50K can't be allocated even if there is 25K + 25K space available.

First Fit:

300K request is allocated from 350K block, 50K is left out.
25K is allocated from the 150K block, 125K is left out.
Then 125K and 50K are allocated to the remaining left out partitions.
So, the first fit can handle requests.

So option B is the correct choice.

Fixed partitioning, also known as static partitioning, is a memory allocation technique used in operating systems to divide the physical memory into fixed-size partitions or regions, each assigned to a specific process or user. Each partition is typically allocated at system boot time and remains dedicated to a specific process until it terminates or releases the partition.

1. In fixed partitioning, the memory is divided into fixed-size chunks, with each chunk being reserved for a specific process. When a process requests memory, the operating system assigns it to the appropriate partition. Each partition is of the same size, and the memory allocation is done at system boot time.
2. Fixed partitioning has several advantages over other memory allocation techniques. First, it is simple and easy to implement. Second, it is predictable, meaning the operating system can ensure a minimum amount of memory for each process. Third, it can prevent processes from interfering with each other's memory space, improving the security and stability of the system.
3. However, fixed partitioning also has some disadvantages. It can lead to internal fragmentation, where memory in a partition remains unused. This can happen when the process's memory requirements are smaller than the partition size, leaving some memory unused. Additionally, fixed partitioning limits the number of processes that can run concurrently, as each process requires a dedicated partition.

Overall, fixed partitioning is a useful memory allocation technique in situations where the number of processes is fixed, and the memory requirements for each process are known in advance. It is commonly used in [embedded systems](#), [real-time systems](#), and systems with limited memory resources.

In operating systems, Memory Management is the function responsible for allocating and managing a computer's main memory. [Memory Management](#) function keeps track of the

status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

There are two Memory Management Techniques:

1. **Contiguous**
2. **Non-Contiguous**

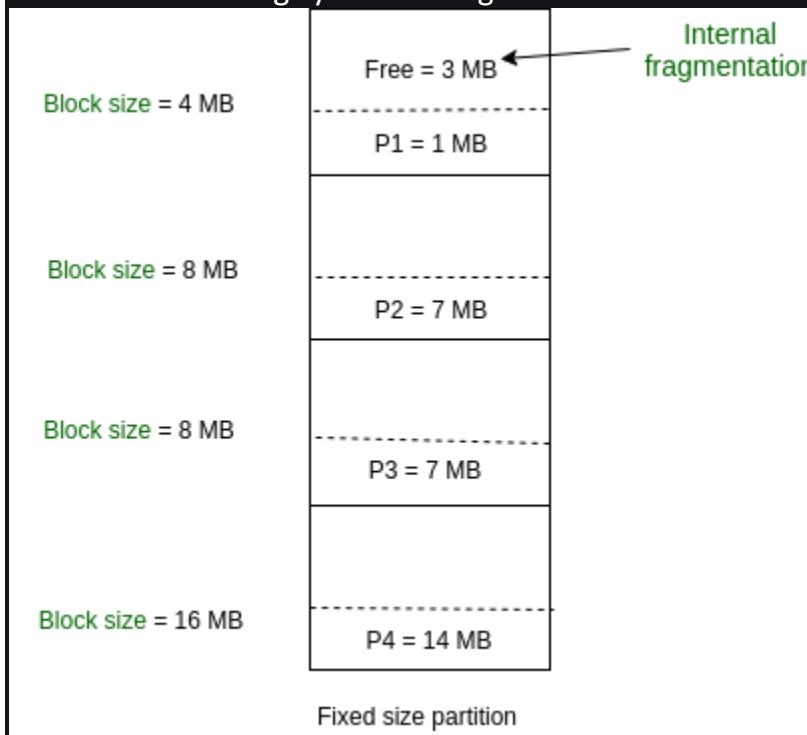
In Contiguous Technique, executing process must be loaded entirely in the main memory.

Contiguous Technique can be divided into:

- Fixed (or static) partitioning
- Variable (or dynamic) partitioning

Fixed Partitioning:

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



As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory.

Hence, Internal Fragmentation in first block is $(4-1) = 3\text{MB}$.

Sum of Internal Fragmentation in every block = $(4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7\text{MB}$.

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

There are some advantages and disadvantages of fixed partitioning.

Advantages of Fixed Partitioning –

- **Easy to implement:** The algorithms needed to implement Fixed Partitioning are straightforward and easy to implement.
- **Low overhead:** Fixed Partitioning requires minimal overhead, which makes it ideal for systems with limited resources.
- **Predictable:** Fixed Partitioning ensures a predictable amount of memory for each process.
- **No external fragmentation:** Fixed Partitioning eliminates the problem of external fragmentation.
- **Suitable for systems with a fixed number of processes:** Fixed Partitioning is well-suited for systems with a fixed number of processes and known memory requirements.
- **Prevents processes from interfering with each other:** Fixed Partitioning ensures that processes do not interfere with each other's memory space.
- **Efficient use of memory:** Fixed Partitioning ensures that memory is used efficiently by allocating it to fixed-sized partitions.
- **Good for batch processing:** Fixed Partitioning is ideal for batch processing environments where the number of processes is fixed.
- **Better control over memory allocation:** Fixed Partitioning gives the operating system better control over the allocation of memory.
- **Easy to debug:** Fixed Partitioning is easy to debug since the size and location of each process are predetermined.

Disadvantages of Fixed Partitioning –

1. Internal Fragmentation:

Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. External Fragmentation:

The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).

3. Limit process size:

Process of size greater than the size of the partition in Main Memory cannot be accommodated. The partition size cannot be varied according to the size of the incoming process size. Hence, the process size of 32MB in the above-stated example is invalid.

4. Limitation on Degree of Multiprogramming:

Partitions in Main Memory are made before execution or during system

configure. Main Memory is divided into a fixed number of partitions. Suppose if there are n partitions in RAM and m are the number of processes, then $n \geq m$ condition must be fulfilled. Number of processes greater than the number of partitions in RAM is invalid in Fixed Partitioning.

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

5.11 Types of Memory Management Techniques

- Contiguous
- Non-Contiguous

In Contiguous Technique, executing process must be loaded entirely in the main memory. The contiguous Technique can be divided into:

1. [Fixed \(or static\) partitioning](#)
2. Variable (or dynamic) partitioning

5.12 Variable (or Dynamic) Partitioning

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

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

Dynamic Partitioning

5.13 Advantages of Variable Partitioning

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

5.14 Disadvantages of Variable Partitioning

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

present in the main memory to get executed. Hence it results in External Fragmentation.

No Internal Fragmentation

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

Learn more

5.15 Key Points On Variable (or Dynamic) Partitioning in Operating Systems

- Variable (or dynamic) partitioning is a memory allocation technique that allows memory partitions to be created and resized dynamically as needed.
- The operating system maintains a table of free memory blocks or holes, each of which represents a potential partition. When a process requests memory, the [operating system](#) searches the table for a suitable hole that can accommodate the requested amount of memory.
- Dynamic partitioning reduces internal fragmentation by allocating memory more efficiently, allows multiple processes to share the same memory space, and is flexible in accommodating processes with varying memory requirements.
- However, dynamic partitioning can also lead to external fragmentation and requires more complex [memory management](#) algorithms, which can make it slower than fixed partitioning.
- Understanding dynamic partitioning is essential for operating system design and implementation, as well as for system-level programming.
- Non-contiguous allocation, also known as dynamic or linked allocation, is a memory allocation technique used in operating systems to allocate memory to processes

that do not require a contiguous block of memory. In this technique, each process is allocated a series of non-contiguous blocks of memory that can be located anywhere in the physical memory.

- Non-contiguous allocation involves the use of pointers to link the non-contiguous memory blocks allocated to a process. These pointers are used by the operating system to keep track of the memory blocks allocated to the process and to locate them during the execution of the process.
- There are several advantages to non-contiguous allocation.
- First, it reduces internal fragmentation since memory blocks can be allocated as needed, regardless of their physical location. Second, it allows processes to be allocated memory in a more flexible and efficient manner since the operating system can allocate memory to a process wherever free memory is available.
- However, non-contiguous allocation also has some disadvantages.
- It can lead to external fragmentation, where the available memory is broken into small, non-contiguous blocks, making it difficult to allocate large blocks of memory to a process. Additionally, the use of pointers to link memory blocks can introduce additional overhead, leading to slower memory allocation and deallocation times.

In operating systems, logical and physical addresses are used to manage and access memory. Here's an overview of each:

Logical address: A logical address, also known as a virtual address, is an address generated by the CPU during program execution. It is the address seen by the process and is relative to the program's address space. The process accesses memory using logical addresses, which are translated by the operating system into physical addresses.

Physical address: A physical address is the actual address in main memory where data is stored. It is a location in physical memory, as opposed to a virtual address. Physical addresses are used by the memory management unit (MMU) to translate logical addresses into physical addresses.

The translation from logical to physical addresses is performed by the operating system's memory management unit. The MMU uses a page table to translate logical addresses into physical addresses. The page table maps each logical page number to a physical frame number.

The similarity between logical and physical addresses in the operating system are listed below:

- Both logical and physical addresses are used to identify a specific location in memory.

- Both logical and physical addresses can be represented in different formats, such as binary, hexadecimal, or decimal.
- Both logical and physical addresses have a finite range, which is determined by the number of bits used to represent them.

Here are some important points about logical and physical addresses in operating systems:

The use of logical addresses provides a layer of abstraction that allows processes to access memory without knowing the physical memory location.

Logical addresses are mapped to physical addresses using a page table. The page table contains information about the mapping between logical and physical addresses.

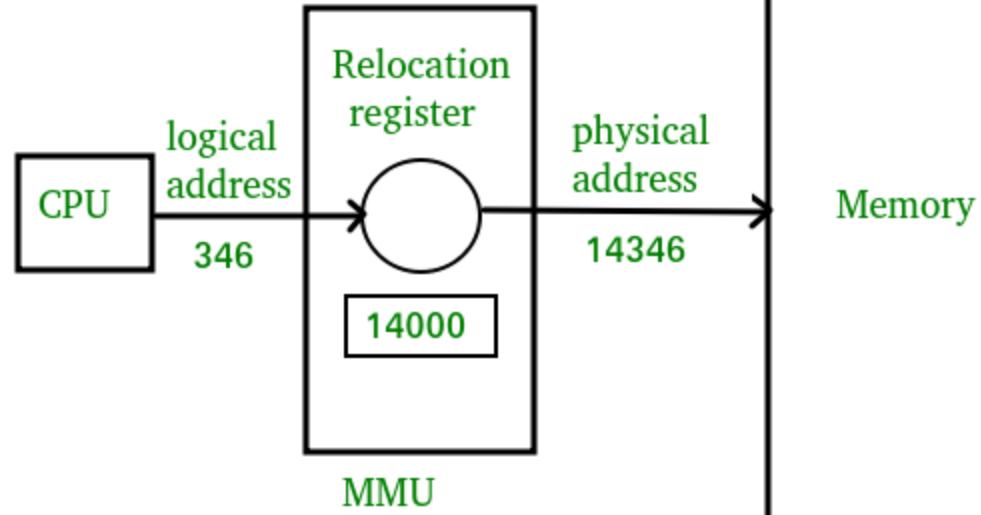
The MMU translates logical addresses into physical addresses using the page table. This translation is transparent to the process and is performed by hardware.

The use of logical and physical addresses allows the operating system to manage memory more efficiently by using techniques such as paging and segmentation.

Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program's perspective.

The hardware device called [Memory-Management](#) Unit is used for mapping logical address to its corresponding physical address.

Physical Address identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.



Mapping virtual-address to physical-addresses

Differences Between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.
3. The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time [address binding](#) method. Please refer [this](#) for details.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

Comparison Chart:

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.

Address Binding :

The Association of program instruction and data to the actual physical memory locations is called the Address Binding. Let's consider the following example given below for better understanding.

Consider a program P1 has the set of instruction such that I1, I2, I3, I4, and program counter value is 10, 20, 30, 40 respectively.

Types of Address Binding :

Address Binding divided into three types as follows.

1. [Compile-time Address Binding](#)
2. [Load time Address Binding](#)

3. Execution time Address Binding

Compile-time Address Binding :

- If the compiler is responsible for performing address binding then it is called compile-time address binding.
- It will be done before loading the program into memory.
- The compiler interacts with an OS memory manager to perform compile-time address binding.

Load time Address Binding :

- It will be done after loading the program into memory.
- This type of address binding will be done by the OS memory manager i.e loader.

Execution time or dynamic Address Binding :

- It will be postponed even after loading the program into memory.
- The program will be kept on changing the locations in memory until the time of program execution.
- The dynamic type of address binding done by the processor at the time of program execution.

Note :

The majority of the Operating System practically implement dynamic loading, dynamic linking, dynamic address binding. For example – Windows, Linux, Unix all popular OS.

Memory Allocation Techniques:

To store the data and to manage the processes, we need a large-sized memory and, at the same time, we need to access the data as fast as possible. But if we increase the size of memory, the access time will also increase and, as we know, the CPU always generates addresses for secondary memory, i.e. logical addresses. But we want to access the main memory, so we need Address translation of logical address into physical address.

The main memory interacts with both the user processes and the operating system. So we need to efficiently use the main memory. Main memory is divided into non-overlapping memory regions called partitions.

The main memory can be broadly allocated in two ways –

1. [Contiguous memory allocation](#)
2. [Non-Contiguous memory allocation](#)

Contiguous memory allocation can be categorized into two ways :

1. [Fixed partition scheme](#)
2. [Variable partition scheme](#).

Different Partition Allocation methods are used in Contiguous memory allocations –

1. [First Fit](#)
2. [Best Fit](#)
3. [Worst Fit](#)
4. [Next Fit](#)

Non-Contiguous memory allocation can be categorized into many ways :

1. [Paging](#)
2. [Multilevel paging](#)

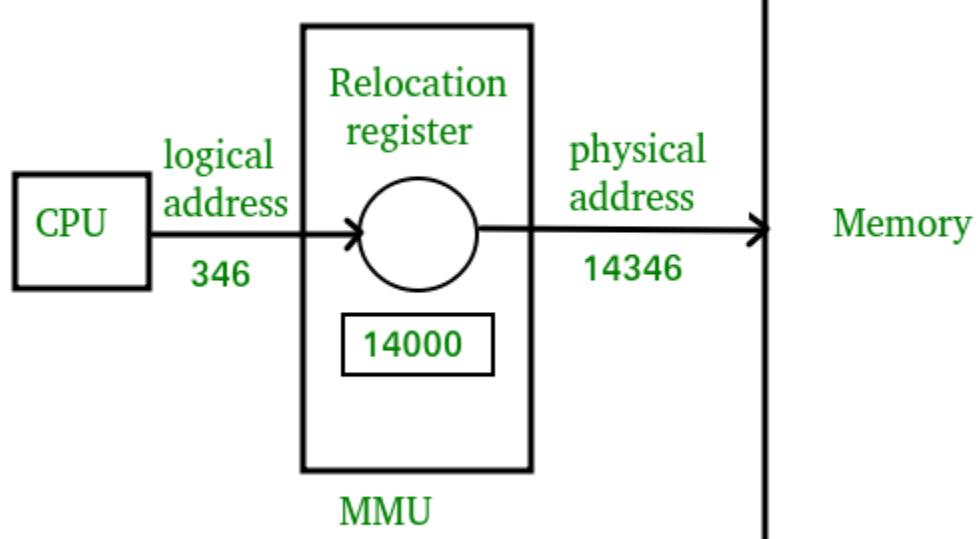
3. [Inverted paging](#)
4. [Segmentation](#)
5. [Segmented paging](#)

MMU(Memory Management Unit) :

The run time mapping between Virtual address and Physical Address is done by a hardware device known as MMU.

In memory management, the Operating System will handle the processes and move the processes between disk and memory for execution . It keeps track of available and used memory.

MMU scheme :



Dynamic relocation using a relocation register.

1. CPU will generate logical address for eg: 346
2. MMU will generate a relocation register (base register) for eg: 14000
3. In memory, the physical address is located eg:(346+14000= 14346)

The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory. The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all like the number 346.

The user program generates only logical addresses. However, these logical addresses must be mapped to physical addresses before they are used.

Address binding :

Address binding is the process of mapping from one address space to another address space. Logical address is an address generated by the CPU during execution, whereas Physical Address refers to the location in the memory unit(the one that is loaded into memory).The logical address undergoes translation by the MMU or address translation unit in particular. The output of this process is the appropriate physical address or the location of code/data in RAM.

An address binding can be done in three different ways :

Compile Time –

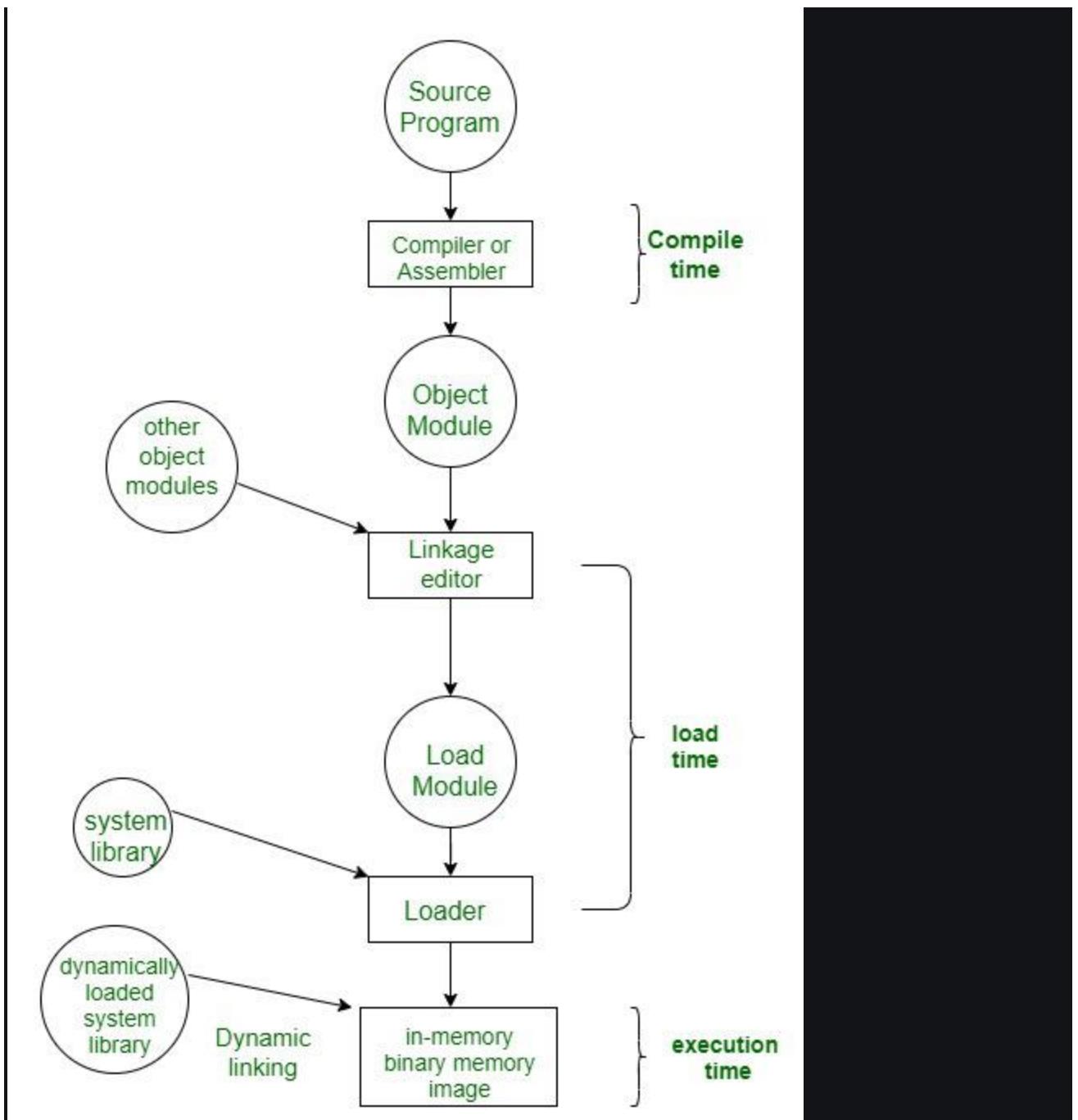
If you know that during compile time, where process will reside in memory, then an absolute address is generated. i.e The physical address is embedded to the executable of the program during compilation. Loading the executable as a process in memory is very fast. But if the generated address space is preoccupied by other processes, then the program crashes and it becomes necessary to recompile the program to change the address space.

Load time –

If it is not known at the compile time where the process will reside, then a relocatable address will be generated. The loader translates the relocatable address to an absolute address. The base address of the process in main memory is added to all logical addresses by the loader to generate an absolute address. In this, if the base address of the process changes, then we need to reload the process again.

Execution time –

The instructions are in memory and are being processed by the CPU. Additional memory may be allocated and/or deallocated at this time. This is used if a process can be moved from one memory to another during execution(dynamic linking-Linking that is done during load or run time). e.g – Compaction.



Mapping Virtual Addresses to Physical Addresses :

In Contiguous memory allocation mapping from virtual addresses to physical addresses is not a difficult task, because if we take a process from secondary memory and copy it to the main memory, the addresses will be stored in a contiguous manner, so if we know the base address of the process, we can find out the next addresses.

The Memory Management Unit is a combination of 2 registers –

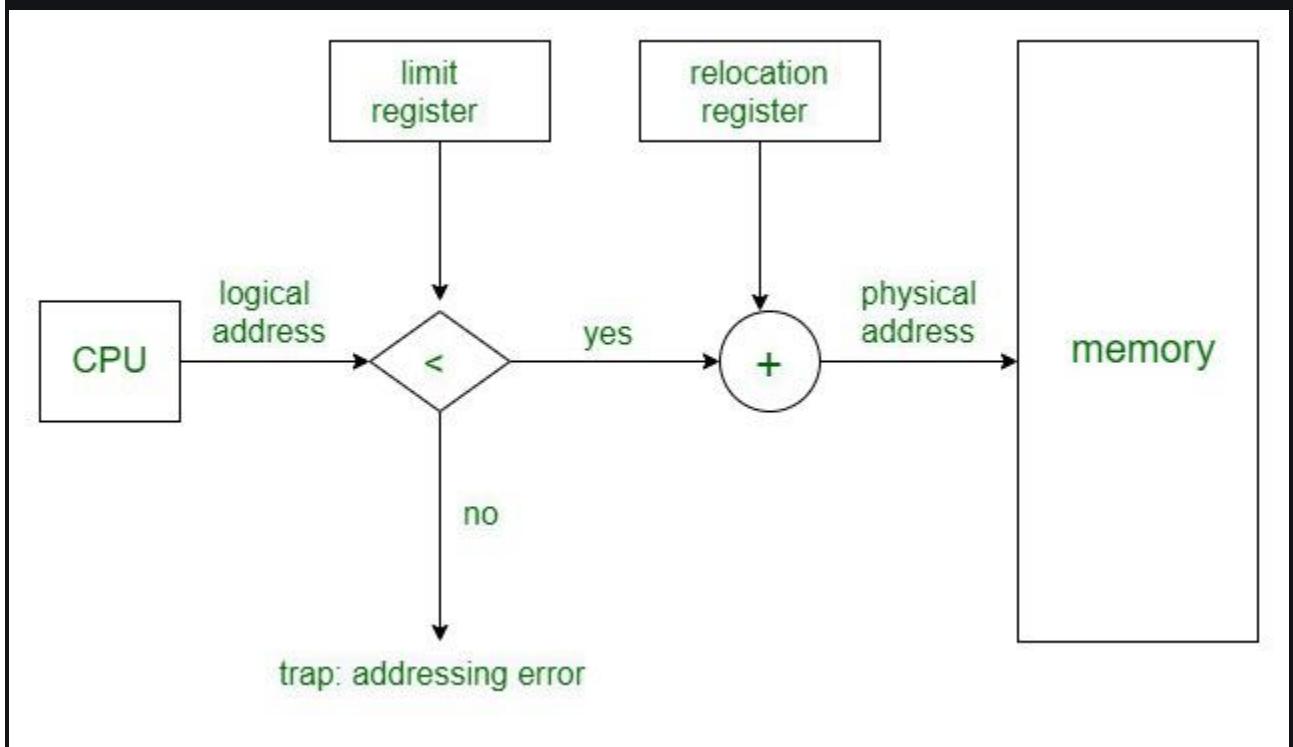
1. Base Register (Relocation Register)
2. Limit Register.

Base Register – contains the starting physical address of the process.

Limit Register -mentions the limit relative to the base address on the region occupied by the process.

The logical address generated by the CPU is first checked by the limit register, If the value of the logical address generated is less than the value of the limit register, the base address stored in the relocation register is added to the logical address to get the physical address of the memory location.

If the logical address value is greater than the limit register, then the CPU traps to the OS, and the OS terminates the program by giving fatal error.



In Non Contiguous Memory allocation, processes can be allocated anywhere in available space. The address translation in non-contiguous memory allocation is difficult.

There are several techniques used for address translation in non contiguous memory allocation like [Paging](#), [Multilevel paging](#), [Inverted paging](#), [Segmentation](#), [Segmented paging](#). Different data structures and hardware support like TLB are required in these techniques.

Paging is a memory management scheme that eliminates the need for a [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. Additionally, frames will be used to split the main memory. This scheme permits the physical address space of a process to be non-contiguous.

In paging, the physical memory is divided into fixed-size blocks called page frames, which are the same size as the pages used by the process. The process's logical address space is also divided into fixed-size blocks called pages, which are the same size as the page frames. When a process requests memory, the operating system allocates one or more page frames to the process and maps the process's logical pages to the physical page frames.

The mapping between logical pages and physical page frames is maintained by the page table, which is used by the memory management unit to translate logical addresses into physical addresses. The page table maps each logical page number to a physical page frame number.

5.16 Terminologies Associated with Memory Control

- **Logical Address or Virtual Address:** This is a deal that is generated through the CPU and used by a technique to get the right of entry to reminiscence. It is known as a logical or digital deal because it isn't always a physical vicinity in memory but an opportunity for a connection with a place inside the device's logical address location.
- **Logical Address Space or Virtual Address Space:** This is the set of all logical addresses generated via a software program. It is normally represented in phrases or bytes and is split into regular-duration pages in a paging scheme.
- **Physical Address:** This is a cope that corresponds to a bodily place in reminiscence. It is the actual cope with this that is available on the memory unit and is used by the memory controller to get admission to the reminiscence.
- **Physical Address Space:** This is the set of all bodily addresses that correspond to the logical addresses inside the way's logical deal with place. It is usually represented in words or bytes and is cut up into fixed-size frames in a paging scheme.

In a paging scheme, the logical deal with the region is cut up into steady-duration pages, and every internet web page is mapped to a corresponding body within the physical deal with the vicinity. The going for walks tool keeps a web internet web page desk for every method, which maps the system's logical addresses to its corresponding bodily addresses. When a method accesses memory, the CPU generates a logical address, that is translated to a bodily address using the net page table. The reminiscence controller then uses the physical cope to get the right of entry to the reminiscence.

Example:

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words ($1\text{ G} = 2^{30}$)
- 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 ($1\text{ M} = 2^{20}$)
- 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 the paging technique.

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical Address Space is also split 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)

Paging

The address generated by the 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 a 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

In a paging scheme, the physical cope with the area is divided into fixed-length frames, each of which contains some bytes or words. When a manner is running, its logical address space is split into constant-size pages, which might be mapped to corresponding frames within the physical address space.

To represent a physical address in this scheme, parts are commonly used:

Frame range: This is the variety of the frame within the physical cope with the area that consists of the byte or phrase being addressed. The wide variety of bits required to represent the body range relies upon the scale of the physical cope with the area and the size of each frame. For instance, if the physical cope with area carries 2^{20} frames and each frame is **4KB (2¹² bytes)** in size, then the frame range could require $20 - 12 = 8$ bits.

Frame offset: This is the wide variety of the byte or word within the body this is being addressed. The number of bits required to represent the frame offset relies upon the size of every frame. For instance, if everybody is 4KB in size, then the frame offset could require 12 bits. So, a physical address in this scheme may be represented as follows:

Physical Address = (Frame Number << Number of Bits in Frame Offset) + Frame Offset, where “<<” represents a bitwise left shift operation.

- 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 the corresponding value is returned.

Paging is a memory management technique used in operating systems to manage memory and allocate memory to processes. In paging, memory is divided into fixed-size blocks called pages, and processes are allocated memory in terms of these pages. Each page is of the same size, and the size is typically a power of 2, such as 4KB or 8 KB.

Multilevel Paging is a paging scheme that consists of two or more levels of page tables in a hierarchical manner. It is also known as hierarchical paging. The entries of the level 1 page table are pointers to a level 2 page table and entries of the level 2 page tables are pointers to a level 3 page table and so on. The entries of the last level page table store actual frame information. Level 1 contains a single-page table and the address of that table is stored in **PTBR (Page Table Base Register)**.

5.17 Why Multi-Level Paging is Required?

Consider a 32-bit physical address space with page size = 4KB and let there be $2^{20} = 1M$, the total entries in the page table, page table entry size = $2^{32}/2^{12} = 2^{20}$, and adding some protection bits and a dirty bit in the page table entry. Now page table size = $2^{20} * 24 = 3MB$ which should be in the physical memory and since each process has its own page table there is so much memory wastage only for storing page tables.

One solution to the large memory requirements of the page tables is to use multilevel paging, only the outermost page table will reside in the main memory and other page tables will be brought to the main memory as per the requirement because, at a particular time, we do not need complete page table, also we can save much memory space because outermost page table can fit in exactly one frame.

Level 1

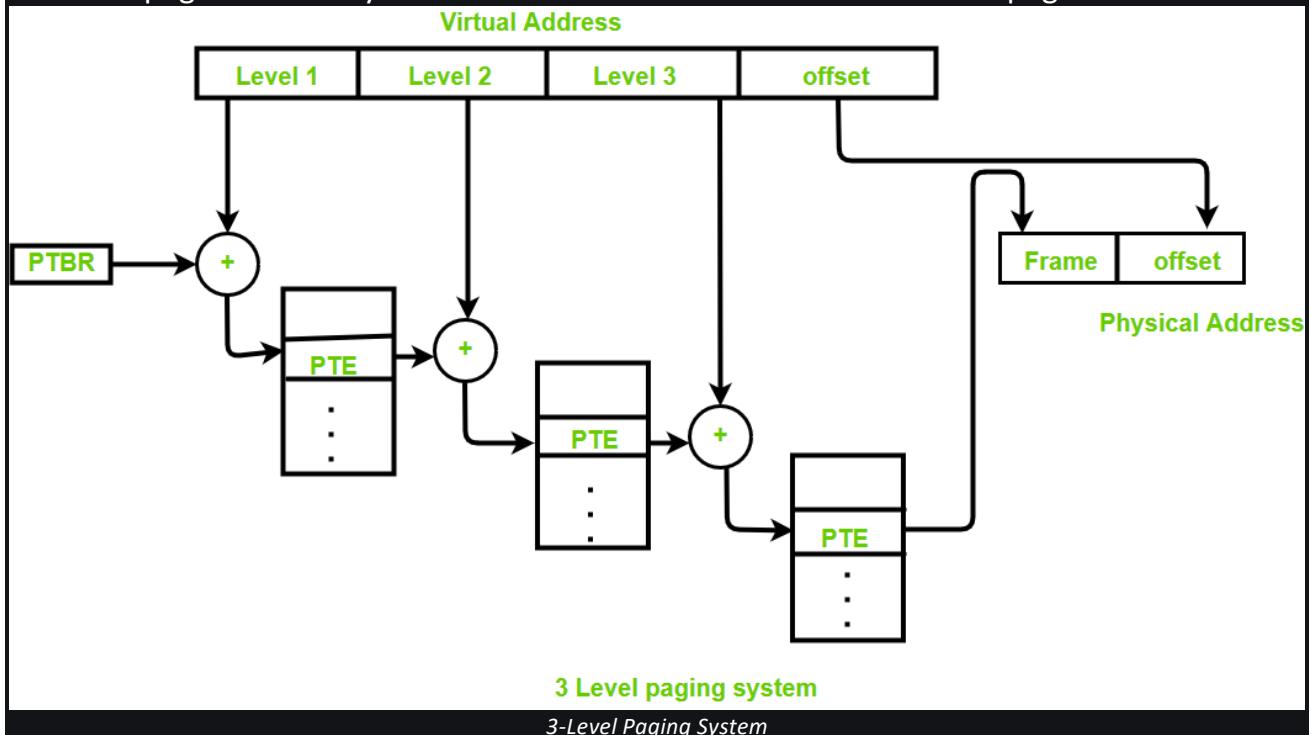
Level 2

.....

Level n

offset

In multilevel paging whatever may be levels of paging, all the page tables will be stored in the main memory. So it requires more than one memory access to get the physical address of the page frame. One access for each level is needed. Each page table entry **except** the last level page table entry contains the base address of the next level page table.



Reference to actual page frame:

- Reference to PTE in level 1 Page Table = PTBR Value + Level 1 Offset present in the Virtual Address.
- Reference to PTE in level 2 Page Table = Base address (present in Level 1 PTE) + Level 2 offset (present in VA).
- Reference to PTE in level 3 Page Table= Base address (present in Level 2 PTE) + Level 3 offset (present in VA).
- Actual page frame address = PTE (present in level 3).

Generally, the page table size will be equal to the size of the page.

Most Operating Systems implement a separate page table for each process, i.e. for the 'n' number of processes running on a [Multiprocessing/ Timesharing Operating System](#), there is an 'n' number of page tables stored in the memory. Sometimes when a process is very large and it occupies virtual memory then with the size of the process, its page table size also increases substantially.

Through this example, it can be concluded that for multiple processes running simultaneously in an OS, a considerable part of memory is occupied by page tables only. Operating Systems also incorporate **multilevel paging schemes** which further increase the space required for storing the page tables and a large amount of memory is invested in storing them. The amount of memory occupied by the page tables can turn out to be a huge overhead and is always unacceptable as main memory is always a scarce resource. Various efforts are made to utilize the memory efficiently and to maintain a good balance in the level of [multiprogramming](#) and efficient CPU utilization.

5.18 What is an Inverted Page Table?

In an operating system that uses virtual memory, an [Inverted Page Table \(IPT\)](#) is a data structure used to map physical memory pages to virtual memory pages. Unlike a traditional Page Table, which is a per-process data structure, an IPT is a system-wide data structure that contains an entry for each physical page in memory.

An alternate approach is to use the **Inverted Page Table** structure that consists of a one-page table entry for every frame of the main memory. So the number of page table entries in the Inverted Page Table reduces to the number of frames in physical memory and a single page table is used to represent the paging information of all the processes.

Through the inverted page table, the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together. This technique is called inverted paging as the indexing is done concerning the frame number instead of the logical page number.

5.19 Components of Inverted Page Table

Each entry in the page table contains the following fields.

- **Page number:** It specifies the page number range of the logical address.
- **Process id:** An inverted page table contains the address space information of all the processes in execution. Since two different processes can have a similar set of virtual addresses, it becomes necessary in the Inverted Page Table to store a process Id of each process to identify its address space uniquely. This is done by using the combination of PID and Page Number. So this Process Id acts as an address space identifier and ensures that a virtual page for a particular process is mapped correctly to the corresponding physical frame.

- **Control bits:** These bits are used to store extra paging-related information. These include the valid bit, dirty bit, reference bits, protection, and locking information bits.
- **Chained pointer:** It may be possible sometimes that two or more processes share a part of the main memory. In this case, two or more logical pages map to the same [Page Table Entry](#) then a chaining pointer is used to map the details of these logical pages to the root page table.

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide. Here the user's view is mapped to physical memory.

5.20 Types of Segmentation in Operating System

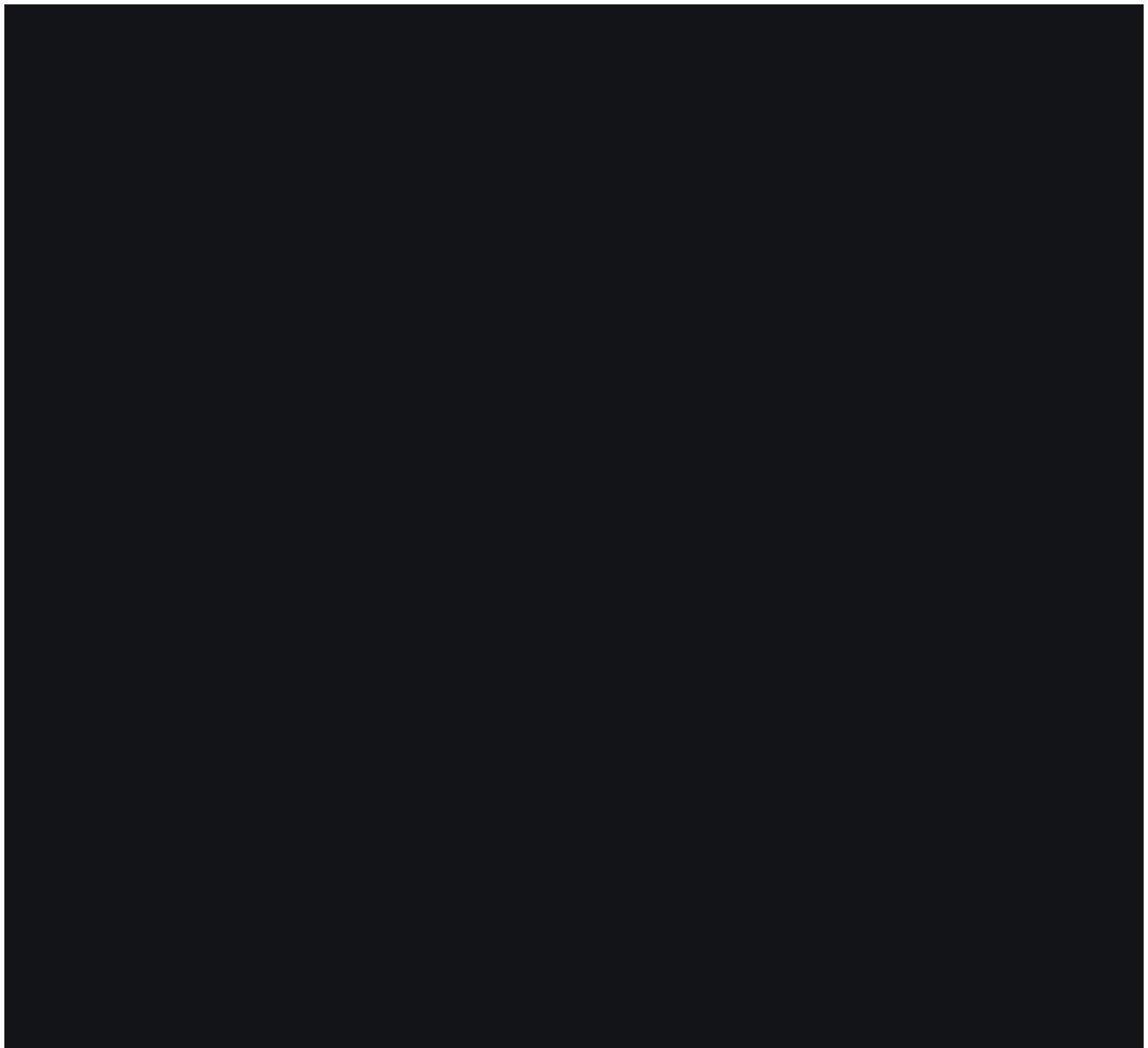
- **Virtual Memory Segmentation:** Each process is divided into a number of segments, but the segmentation is not done all at once. This segmentation may or may not take place at the run time of the program.
- **Simple Segmentation:** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

5.21 What is Segment Table?

It maps a two-dimensional Logical address into a one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Segment Limit:** Also known as segment offset. It specifies the length of the segment.



Translation

The address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.

5.22 Advantages of Segmentation in Operating System

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.
- As a complete module is loaded all at once, segmentation improves CPU utilization.
- The user's perception of physical memory is quite similar to segmentation. Users can divide user programs into modules via segmentation. These modules are nothing more than separate processes' codes.
- The user specifies the segment size, whereas, in paging, the hardware determines the page size.
- Segmentation is a method that can be used to segregate data from security operations.
- **Flexibility:** Segmentation provides a higher degree of flexibility than paging. Segments can be of variable size, and processes can be designed to have multiple segments, allowing for more fine-grained memory allocation.

- **Sharing:** Segmentation allows for sharing of memory segments between processes. This can be useful for inter-process communication or for sharing code libraries.
- **Protection:** Segmentation provides a level of protection between segments, preventing one process from accessing or modifying another process's memory segment. This can help increase the security and stability of the system.

5.23 Disadvantages of Segmentation in Operating System

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.
- Overhead is associated with keeping a segment table for each activity.
- Due to the need for two memory accesses, one for the segment table and the other for main memory, access time to retrieve the instruction increases.
- **Fragmentation:** As mentioned, segmentation can lead to external fragmentation as memory becomes divided into smaller segments. This can lead to wasted memory and decreased performance.
- **Overhead:** Using a segment table can increase overhead and reduce performance. Each segment table entry requires additional memory, and accessing the table to retrieve memory locations can increase the time needed for memory operations.
- **Complexity:** Segmentation can be more complex to implement and manage than paging. In particular, managing multiple segments per process can be challenging, and the potential for segmentation faults can increase as a result.

Paged Segmentation and Segmented Paging are two different memory management techniques that combine the benefits of paging and segmentation.

1. Paged Segmentation is a memory management technique that divides a process's address space into segments and then divides each segment into pages. This allows for a flexible allocation of memory, where each segment can have a different size, and each page can have a different size within a segment.
2. Segmented Paging, on the other hand, is a memory management technique that divides the physical memory into pages, and then maps each logical address used by a process to a physical page. In this approach, segments are used to map virtual memory addresses to physical memory addresses, rather than dividing the virtual memory into pages.
3. Both Paged Segmentation and Segmented Paging provide the benefits of paging, such as improved memory utilization, reduced fragmentation, and increased performance. They also provide the benefits of segmentation, such as increased flexibility in memory allocation, improved protection and security, and reduced overhead in memory management.

However, both techniques can also introduce additional complexity and overhead in the memory management process. The choice between Paged Segmentation and Segmented

Paging depends on the specific requirements and constraints of a system, and often requires trade-offs between flexibility, performance, and overhead.

Major Limitation of Single Level Paging

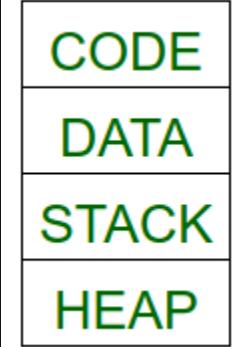
A big challenge with single level paging is that if the logical address space is large, then the page table may take up a lot of space in main memory. For instance, consider that logical address is 32 bit and each page is 4 KB, the number of pages will be 2^{20} pages. The page table without additional bits will be of the size $20 \text{ bits} * 2^{20}$ or 2.5 MB. Since each process has its own page table, a lot of memory will be consumed when single level paging is used. For a system with 64-bit logical address even a page table of single process will not fit in main memory. For a process with a large logical address space, a lot of its page table entries are invalid as a lot of the logical address space goes unused.

Frame No.	Valid
12	Y
29	Y
-	N
-	N
-	N
36	Y
50	Y
-	N
-	N
-	N
-	N
-	N
-	N
14	Y
32	Y
43	Y

Page table with invalid entries

Segmented Paging

A solution to the problem is to use segmentation along with paging to reduce the size of page table. Traditionally, a program is divided into four segments, namely code segment, data segment, stack segment and heap segment.

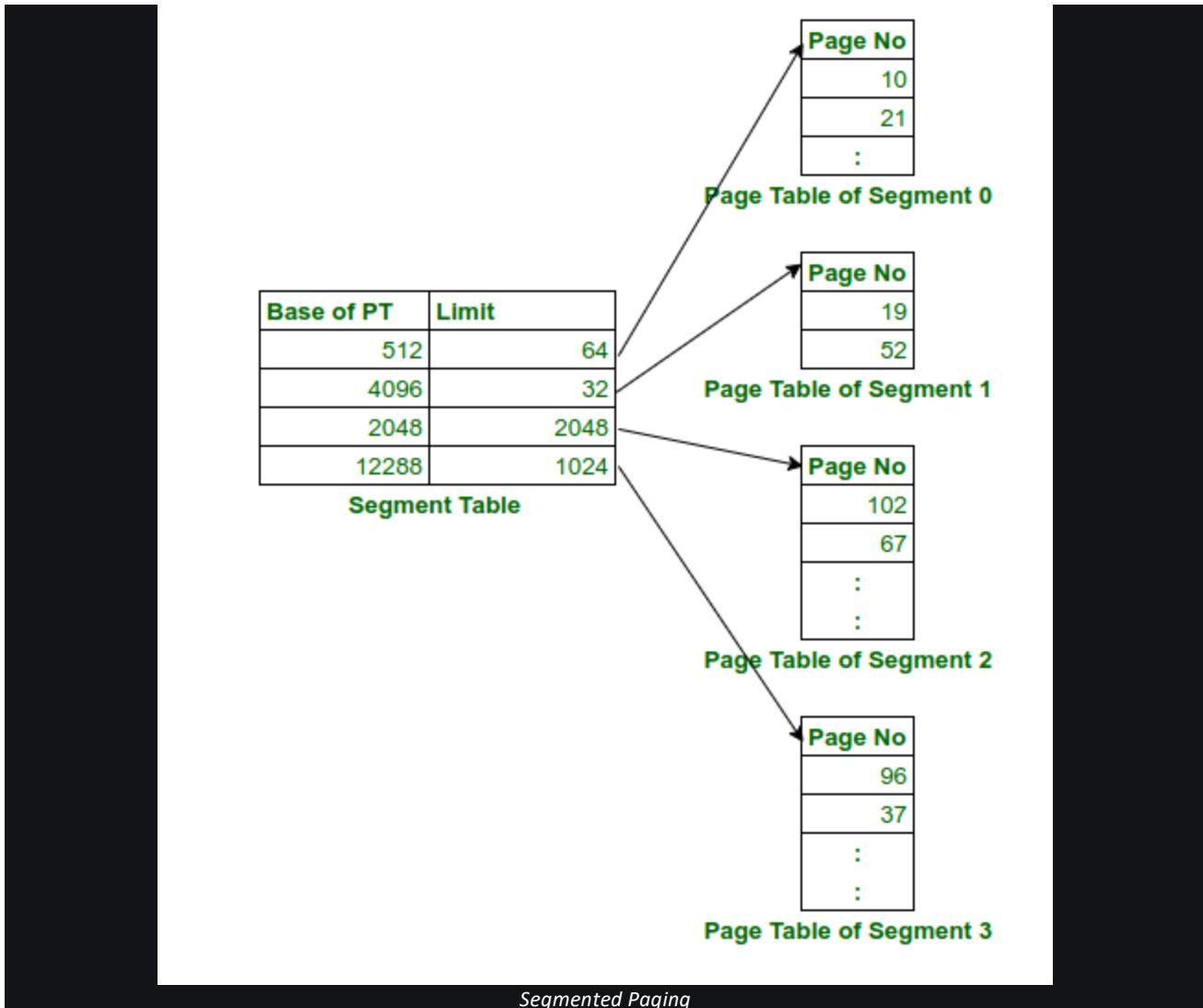


Segments of a process

The size of the page table can be reduced by creating a page table for each segment. To accomplish this hardware support is required. The address provided by CPU will now be partitioned into segment no., page no. and offset.

31 30	29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Seg No	Page No	Offset

The memory management unit (MMU) will use the segment table which will contain the address of page table(base) and limit. The page table will point to the page frames of the segments in main memory.



Advantages of Segmented Paging

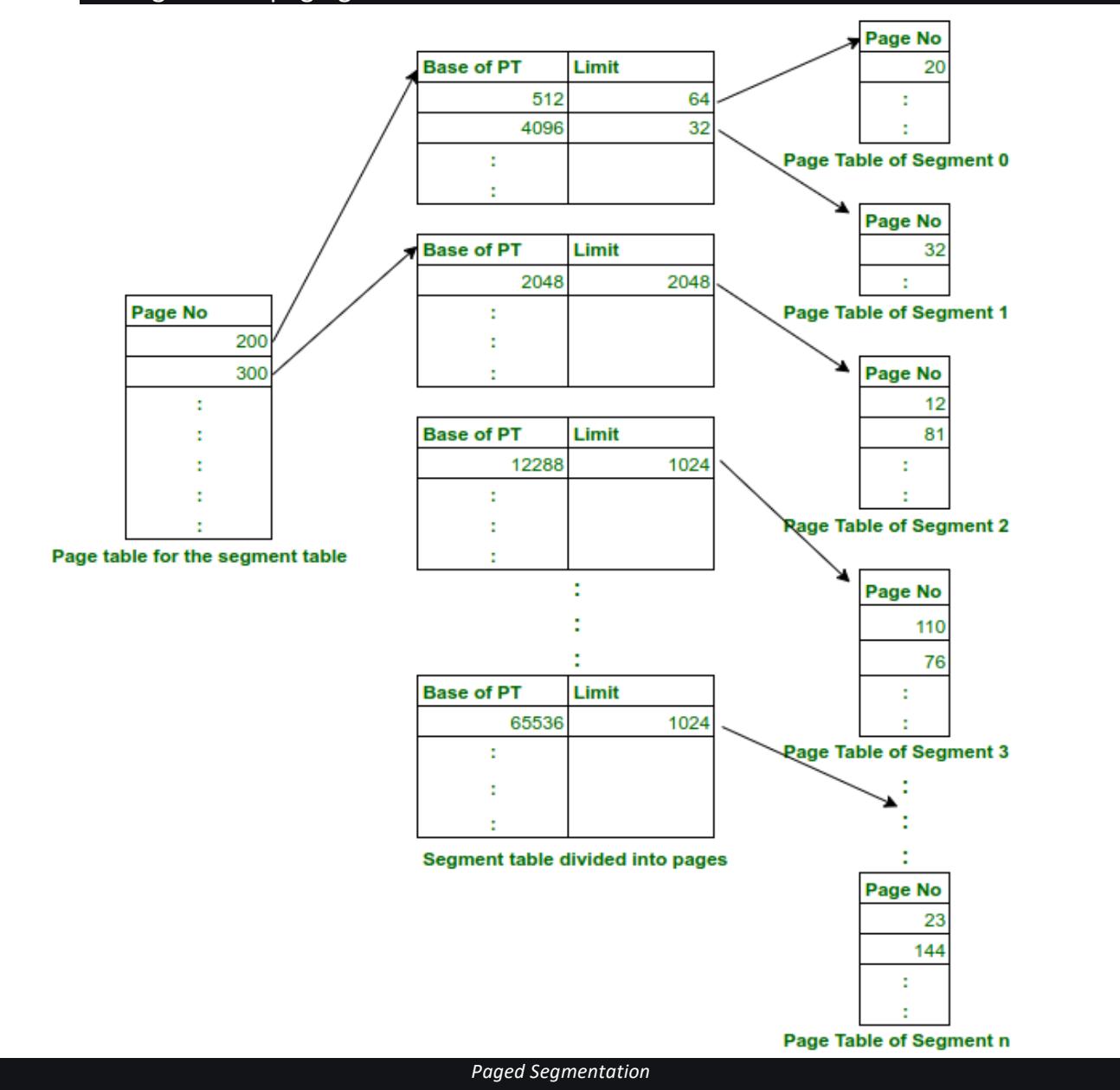
1. The page table size is reduced as pages are present only for data of segments, hence reducing the memory requirements.
2. Gives a programmers view along with the advantages of paging.
3. Reduces external fragmentation in comparison with segmentation.
4. Since the entire segment need not be swapped out, the swapping out into virtual memory becomes easier .

Disadvantages of Segmented Paging

1. Internal fragmentation still exists in pages.
2. Extra hardware is required
3. Translation becomes more sequential increasing the memory access time.
4. External fragmentation occurs because of varying sizes of page tables and varying sizes of segment tables in today's systems.

Paged Segmentation

1. In segmented paging, not every process has the same number of segments and the segment tables can be large in size which will cause external fragmentation due to the varying segment table sizes. To solve this problem, we use **paged segmentation** which requires the segment table to be paged. The logical address generated by the CPU will now consist of page no #1, segment no, page no #2 and offset.
2. The page table even with segmented paging can have a lot of invalid pages. Instead of using multi level paging along with segmented paging, the problem of larger page table can be solved by directly applying multi level paging instead of segmented paging.



Paged Segmentation

Advantages of Paged Segmentation

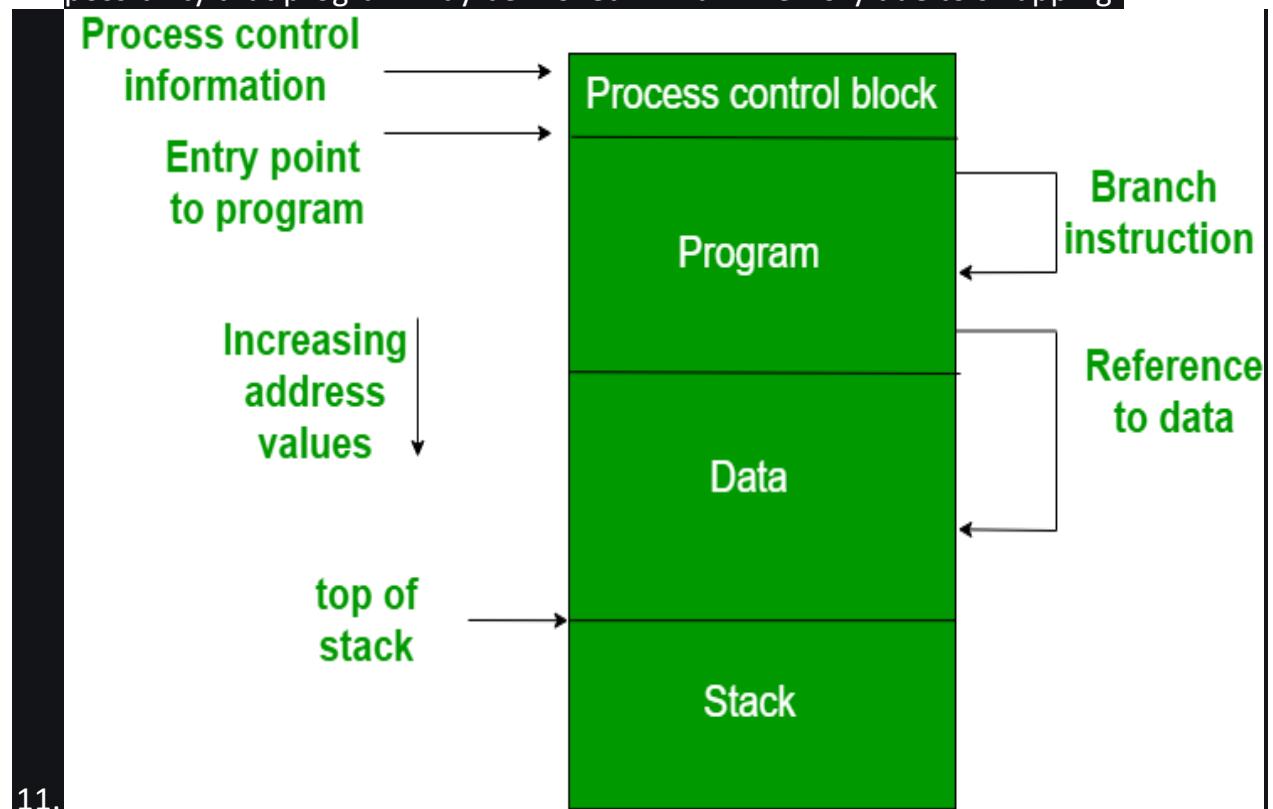
1. No external fragmentation
2. Reduced memory requirements as no. of pages limited to segment size.
3. Page table size is smaller just like segmented paging,
4. Similar to segmented paging, the entire segment need not be swapped out.
5. Increased flexibility in memory allocation: Paged Segmentation allows for a flexible allocation of memory, where each segment can have a different size, and each page can have a different size within a segment.
6. Improved protection and security: Paged Segmentation provides better protection and security by isolating each segment and its pages, preventing a single segment from affecting the entire process's memory.
Increased program structure: Paged Segmentation provides a natural program structure, with each segment representing a different logical part of a program.
7. Improved error detection and recovery: Paged Segmentation enables the detection of memory errors and the recovery of individual segments, rather than the entire process's memory.
8. Reduced overhead in memory management: Paged Segmentation reduces the overhead in memory management by eliminating the need to maintain a single, large page table for the entire process's memory.
9. Improved memory utilization: Paged Segmentation can improve memory utilization by reducing fragmentation and allowing for the allocation of larger blocks of contiguous memory to each segment.

Disadvantages of Paged Segmentation

1. Internal fragmentation remains a problem.
2. Hardware is complexer than segmented paging.
3. Extra level of paging at first stage adds to the delay in memory access.
4. Increased complexity in memory management: Paged Segmentation introduces additional complexity in the memory management process, as it requires the maintenance of multiple page tables for each segment, rather than a single page table for the entire process's memory.
5. Increased overhead in memory access: Paged Segmentation introduces additional overhead in memory access, as it requires multiple lookups in multiple page tables to access a single memory location.
6. Reduced performance: Paged Segmentation can result in reduced performance, as the additional overhead in memory management and access can slow down the overall process.
7. Increased storage overhead: Paged Segmentation requires additional storage overhead, as it requires additional data structures to store the multiple page tables for each segment.
8. Increased code size: Paged Segmentation can result in increased code size, as the additional code required to manage the multiple page tables can take up valuable memory space.

9. Reduced address space: Paged Segmentation can result in a reduced address space, as some of the available memory must be reserved for the storage of the multiple page tables.
10. **Relocation** – The available memory is generally shared among a number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of this program. Swapping the active processes in and out of the main memory enables the operating system to have a larger pool of ready-to-execute process.

When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to **relocate** the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.



- 11.
12. The figure depicts a process image. The process image is occupying a continuous region of main memory. The operating system will need to know many things including the location of process control information, the execution stack, and the code entry. Within a program, there are memory references in various instructions and these are called logical addresses.

After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch

instructions contain the address of the next instruction to be executed. Data reference instructions contain the address of byte or word of data referenced.

13. **Protection** – There is always a danger when we have multiple programs at the same time as one program may write to the address space of another program. So every process must be protected against unwanted interference when other process tries to write in a process whether accidental or incidental. Between relocation and protection requirement a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement.

Prediction of the location of a program in main memory is not possible, that's why it is impossible to check the absolute address at compile time to assure protection. Most of the programming language allows the dynamic calculation of address at run time. The memory protection requirement must be satisfied by the processor rather than the operating system because the operating system can hardly control a process when it occupies the processor. Thus it is possible to check the validity of memory references.

14. **Sharing** – A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than have their own separate copy has an advantage.

For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it shared by those processes. It is the task of Memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation supported sharing capabilities.

15. **Logical organization** – Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:

1. Modules are written and compiled independently and all the references from one module to another module are resolved by 'the system at run time.'
2. Different modules are provided with different degrees of protection.
3. There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.

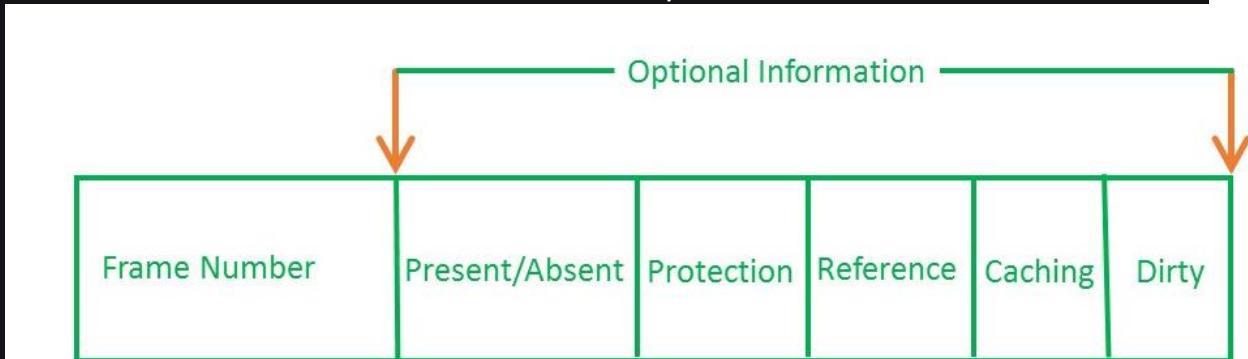
16. Physical organization – The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons:

1. The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.
2. In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

17. A **Page Table** is a data structure used by the operating system to keep track of the mapping between virtual addresses used by a process and the corresponding physical addresses in the system's memory.

18. A Page Table Entry (PTE) is an entry in the Page Table that stores information about a particular page of memory. Each PTE contains information such as the physical address of the page in memory, whether the page is present in memory or not, whether it is writable or not, and other access permissions.

19. The size and format of a PTE can vary depending on the architecture of the system and the operating system used. In general, a PTE contains enough information to allow the operating system to manage memory efficiently and protect the system from malicious or accidental access to memory.



PAGE TABLE ENTRY

20.

21. Page Table Entry

22. The number of Page Table Entries in a Page Table depends on the size of the virtual address space used by a process and the size of the memory pages used by the

system. For example, if the virtual address space of a process is 32 bits, and the system uses 4 KB pages, then the Page Table will have 2^{20} (1 million) entries, with each entry being 4 bytes in size.

In modern systems with large virtual address spaces, such as 64-bit systems, the Page Table can become very large, leading to performance issues. To address this, some systems use hierarchical page tables, where the Page Table is divided into smaller tables, each pointing to a larger table. This allows for more efficient memory management and faster access to Page Table Entries.

5.24 Information Stored in Page Table Entry

- **Frame Number** – It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames. Frame bit is also known as address translation bit.
- **Present/Absent Bit:** Present or absent bit says whether a particular page you are looking for is present or absent. In case it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control [page faults](#) by the operating system to support virtual memory. Sometimes this bit is also known as a **valid/invalid** bit.
- **Protection Bit:** The protection bit says what kind of protection you want on that page. So, these bits are for the protection of the [page frame](#) (read, write, etc).
- **Referenced Bit:** Referenced bit will say whether this page has been referred to in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.
- **Caching Enabled/Disabled:** Sometimes we need fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user. Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of memory. The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information to be consistent, which means whatever information the user has given, the CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit **enables or disables** caching of the page.
- **Modified Bit:** Modified bit says whether the page has been modified or not. Modified means sometimes you might try to write something onto the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. It is set to 1 by hardware on the

write-access to a page which is used to avoid writing when swapped out.

Sometimes this modified bit is also called the **Dirty bit**.

5.25 Advantages of Using a Page Table in a Virtual Memory System

- **Efficient Use of Memory:** [Virtual memory](#) allows the operating system to allocate only the necessary amount of physical memory needed by a process, which reduces memory waste and increases overall system performance.
- **Protection:** Page Tables allow the operating system to control access to memory and protect sensitive data from unauthorized access. Each PTE can be configured with access permissions, such as read-only or no access, to prevent accidental or malicious modification of memory.
- **Flexibility:** Virtual memory allows multiple processes to share the same physical memory space, which increases system flexibility and allows for better resource utilization.
- **Address Translation:** Page Tables provide the mechanism for translating [virtual addresses](#) used by a process into physical addresses in memory, which allows for efficient use of memory and simplifies memory management.
- **Hierarchical Design:** Some systems use hierarchical page tables, which provide a more efficient method for managing large virtual address spaces. Hierarchical page tables divide the Page Table into smaller tables, each pointing to a larger table, which allows for faster access to Page Table Entries and reduces the overall size of the Page Table.

Virtual Memory is a storage allocation scheme in which [secondary memory](#) can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites and program-generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory available not by the actual number of main storage locations.

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

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

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

5.26 Demand Paging

The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging. The process includes the following steps are as follows:

Demand Paging

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

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

5.27 Advantages of Virtual Memory

- **More processes may be maintained in the main memory:** Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- **A process may be larger than all of the main memory:** One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.
- It has twice the capacity for addresses as main memory.
- It makes it possible to run more applications at once.
- Users are spared from having to add memory modules when [RAM](#) space runs out, and applications are liberated from shared memory management.
- When only a portion of a program is required for execution, speed has increased.
- Memory isolation has increased security.
- It makes it possible for several larger applications to run at once.
- Memory allocation is comparatively cheap.
- It doesn't require outside fragmentation.
- It is efficient to manage logical partition workloads using the CPU.
- Automatic data movement is possible.

5.28 Disadvantages of Virtual Memory

- It can slow down the system performance, as data needs to be constantly transferred between the physical memory and the hard disk.
- It can increase the risk of data loss or corruption, as data can be lost if the hard disk fails or if there is a power outage while data is being transferred to or from the hard disk.
- It can increase the complexity of the memory management system, as the operating system needs to manage both physical and virtual memory.

Page Fault Service Time: The time taken to service the page fault is called page fault service time. The page fault service time includes the time taken to perform all the above six steps.

5.29 Swapping

Swapping is a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to the main memory. When a process is busy swapping pages in and out then this situation is called thrashing.

Swapping

5.30 Thrashing

At any given time, only a few pages of any process are in the main memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of the main memory will be occupied with process pages, so that the processor and OS have direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

In the given diagram, the initial degree of multiprogramming up to some extent of point(λ), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multiprogramming the CPU utilization will

drastically fall down and the system will spend more time only on the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called thrashing.

5.31 Causes of Thrashing

1. High Degree of Multiprogramming: If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, fewer frames will be available for each process. Due to this, a [page fault](#) will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:

Let	free	frames	=	400
Case 1:	Number	of	processes	=

Then, each process will get 4 frames.

Case 2: Number of processes = 400

Each process will get 1 frame.

Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed just by swapping pages.

2. Lacks of Frames: If a process has fewer frames then fewer pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence a sufficient amount of frames must be allocated to each process in order to prevent thrashing.

5.32 Recovery of Thrashing

- Do not allow the system to go into thrashing by instructing the long-term scheduler not to bring the processes into memory after the threshold.
- If the system is already thrashing then instruct the [mid-term scheduler](#) to suspend some of the processes so that we can recover the system from thrashing.

5.33 Performance in Virtual Memory

- Let p be the page fault rate($0 \leq p \leq 1$).
- if $p = 0$ no page faults
- if $p = 1$, every reference is a fault.

The performance of a virtual memory management system depends on the total number of page faults, which depend on “[paging policies](#)” and “[frame allocation](#)”

5.34 Frame Allocation

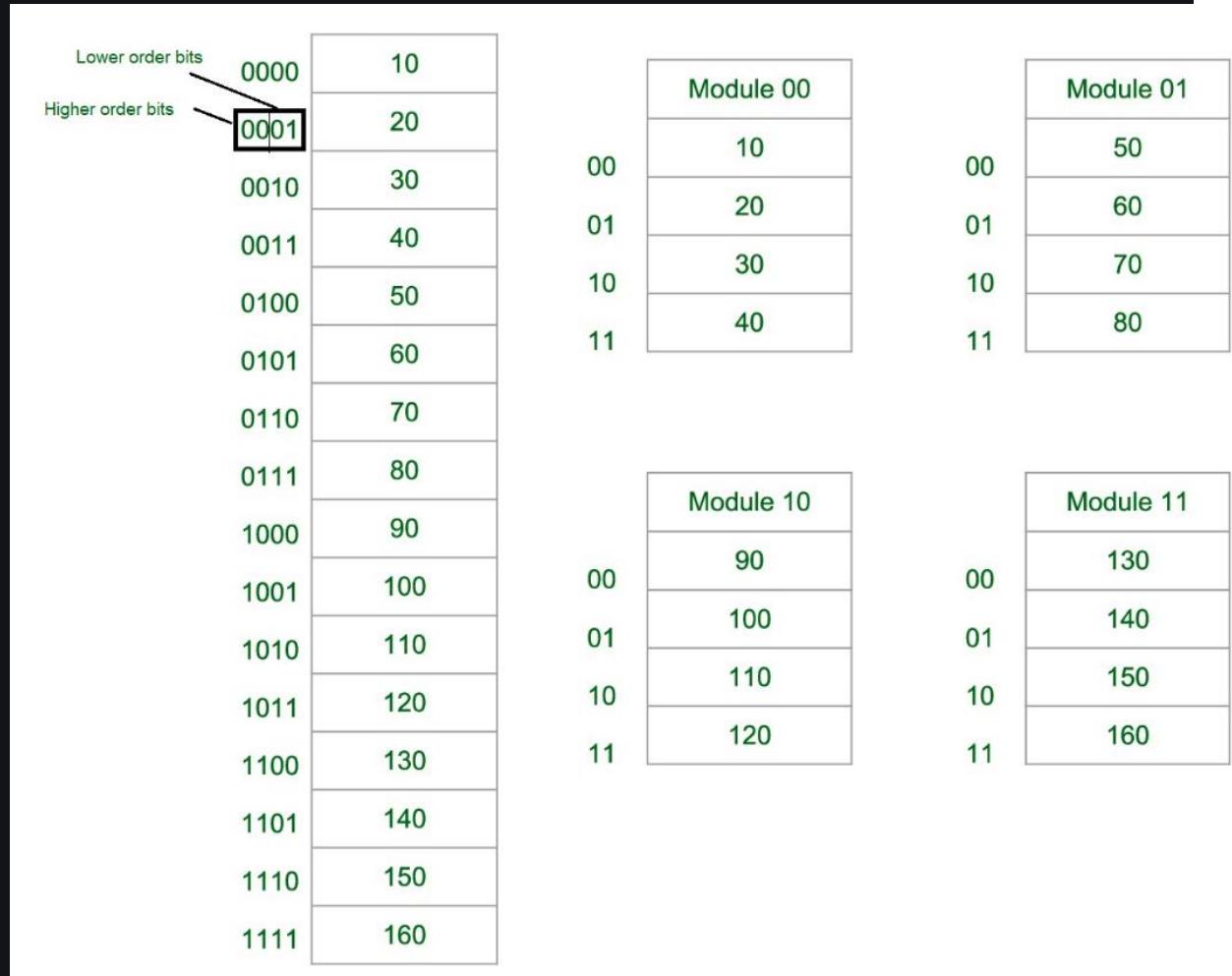
A number of frames allocated to each process in either static or dynamic.

- **Static Allocation:** The number of frame allocations to a process is fixed.

- **Dynamic Allocation:** The number of frames allocated to a process changes.

Paging Policies

- **Fetch Policy:** It decides when a page should be loaded into memory.
- **Replacement Policy:** It decides which page in memory should be replaced.
- **Placement Policy:** It decides where in memory should a page be loaded.
- Abstraction is one of the most important aspects of computing. It is a widely implemented Practice in the Computational field.
- **Memory Interleaving** is less or More an Abstraction technique. Though it's a bit different from Abstraction. It is a Technique that divides memory into a number of modules such that Successive words in the address space are placed in the Different modules.
- **Consecutive Word in a Module:**
-



- **Figure-1:** Consecutive Word in a Module

- Let us assume 16 Data's to be Transferred to the Four Module. Where Module 00 be Module 1, Module 01 be Module 2, Module 10 be Module 3 & Module 11 be Module 4. Also, 10, 20, 30....130 are the data to be transferred.
- From the figure above in Module 1, 10 [Data] is transferred then 20, 30 & finally, 40 which are the Data. That means the data are added consecutively in the Module till its max capacity.
- Most significant bit (MSB) provides the Address of the Module & the least significant bit (LSB) provides the address of the data in the module.
- For **Example**, to get 90 (Data) 1000 will be provided by the processor. This 10 will indicate that the data is in module 10 (module 3) & 00 is the address of 90 in Module 10 (module 3). So,

- **Consecutive Word in Consecutive Module:**
-

	Module 00	Module 01	Module 10	Module 11
00	10	20	30	40
01	50	60	70	80
10	90	100	110	120
11	130	140	150	160

- **Figure-2:** Consecutive Word in Consecutive Module
- Now again we assume 16 Data's to be transferred to the Four Module. But Now the consecutive Data are added in Consecutive Module. That is, 10 [Data] is added in Module 1, 20 [Data] in Module 2 and So on.
- Least Significant Bit (LSB) provides the Address of the Module & Most significant bit (MSB) provides the address of the data in the module.
- For Example, to get 90 (Data) 1000 will be provided by the processor. This 00 will indicate that the data is in module 00 (module 1) & 10 is the address of 90 in Module 00 (module 1). That is,

- **Why do we use Memory Interleaving? [Advantages]:**

Whenever Processor requests Data from the main memory. A block (chunk) of Data is Transferred to the cache and then to Processor. So whenever a cache miss occurs the Data is to be fetched from the main memory. But main memory is relatively

slower than the cache. So to improve the access time of the main memory interleaving is used.

- We can access all four Modules at the same time thus achieving Parallelism. From Figure 2 the data can be acquired from the Module using the Higher bits. This method Uses memory effectively.

Page Table in DBMS:

The operating system is responsible for maintaining a Page Table for each process. We didn't need to record any process identifiers in the Page Table because each process has its own Page Table. A Page Table transforms a specified virtual or logical page number into a real physical frame or page address.

The connection between virtual addresses and physical addresses is stored in a Page Table, a crucial component of a virtual memory system. The crucial component of the virtual address conversion Page Table is needed to get the data in the system's memory.

Inverted Page Table in DBMS:

In an Inverted Page Table, there is a virtual page with respect to each frame that is present in physical memory. As with the normal Page Table, the actual address translation starts with a virtual page and ends with a physical memory frame, but this Page Table is in reverse form because we see from physical memory to virtual memory in the case of mapping

A computer has a sufficient amount of physical memory but most of the time we need more so we swap some memory on disk. Swap space is a space on a hard disk that is a substitute for physical memory. It is used as virtual memory which contains process memory images. Whenever our computer runs short of physical memory it uses its virtual memory and stores information in memory on disk. Swap space helps the computer's operating system in pretending that it has more RAM than it actually has. It is also called a swap file. This interchange of data between virtual memory and real memory is called swapping and space on disk as "swap space".

Swap space, also known as virtual memory or paging space, is a feature of an operating system that allows it to temporarily move inactive or less frequently used pages of memory from RAM to a designated area on the hard disk. The purpose of swap space is to free up physical memory (RAM) so that it can be used for more important tasks, while still allowing the system to maintain the illusion of having more memory than it actually has.

When a system runs out of physical memory, it will start moving pages of memory to the swap space, which is typically a dedicated partition on the hard disk. The pages that are moved to swap space are those that have not been accessed in a while, or are not essential to the current operation of the system.

While swap space can help to avoid system crashes due to lack of memory, it can also have a negative impact on system performance. When pages are swapped to and from disk, it can cause disk I/O operations that slow down the system. Additionally, if the system uses too much swap space, it can lead to thrashing, which is a condition where the system spends most of its time moving pages to and from disk instead of performing useful work.

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens :

- The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
- An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
- Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Some times hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
- Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
- If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
- If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
- As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.
- Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
- Assembly Routine reloads register and other state information, returns to user space to continue execution.

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide. Here the user's view is mapped to physical memory.

5.35 Types of Segmentation in Operating System

- **Virtual Memory Segmentation:** Each process is divided into a number of segments, but the segmentation is not done all at once. This segmentation may or may not take place at the run time of the program.

- **Simple Segmentation:** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

5.36 What is Segment Table?

It maps a two-dimensional Logical address into a one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Segment Limit:** Also known as segment offset. It specifies the length of the segment.

Translation of Two-dimensional Logical Address to Dimensional Physical Address.

In Operating System (Memory Management Technique: [Paging](#)), for each process page table will be created, which will contain a [Page Table Entry \(PTE\)](#). This PTE will contain information like frame number (The address of the main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit, etc). This page table entry ([PTE](#)) will tell where in the main memory the actual page is residing.

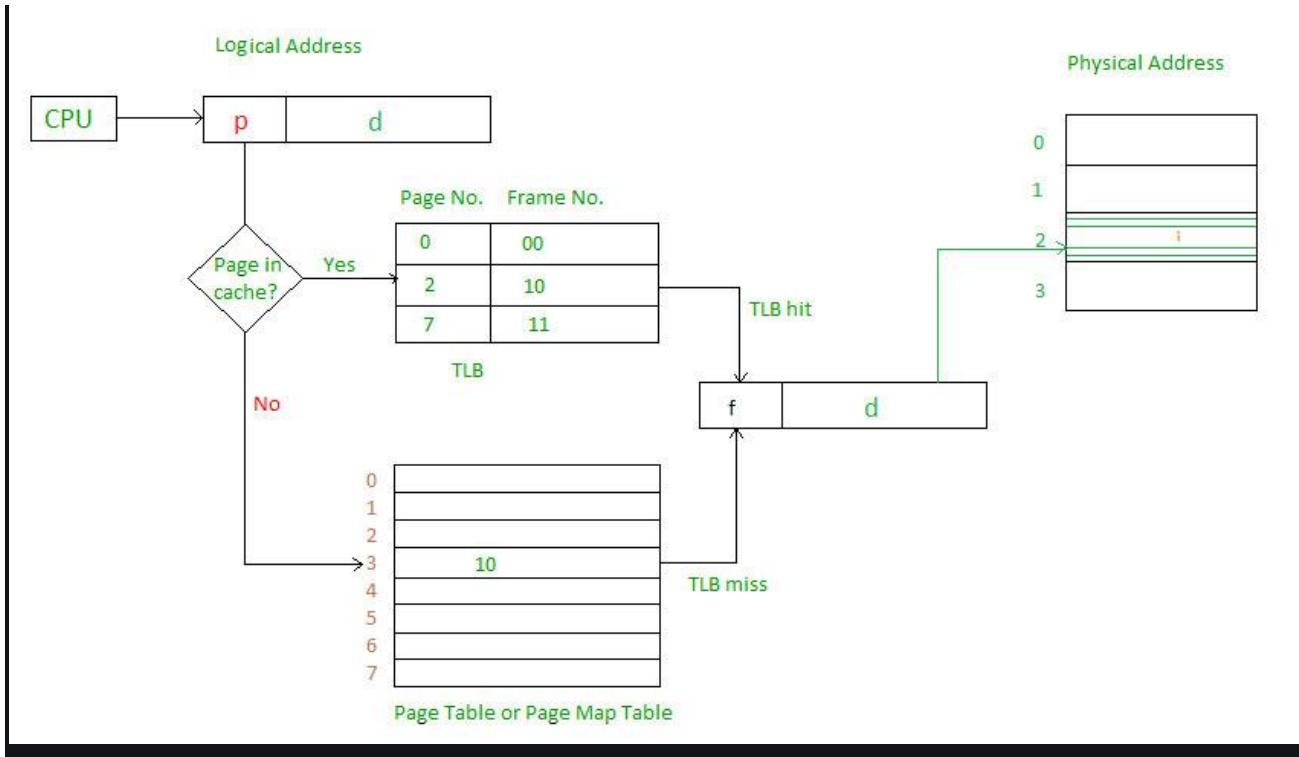
Now the question is where to place the page table, such that overall access time (or reference time) will be less. The problem initially was to fast access the main memory content based on the address generated by the CPU (i.e. [logical/virtual address](#)). Initially, some people thought of using registers to store page tables, as they are high-speed memory so access time will be less.

The idea used here is, to place the page table entries in registers, for each request generated from the CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem is registered size is small (in practice, it can accommodate a maximum of 0.5k to 1k page table entries) and the process size may be big hence the required page table will also be big (let's say this page table contains 1M entries), so registers may not hold all the PTE's of the Page table. So this is not a practical approach.

The entire page table was kept in the main memory to overcome this size issue. but the problem here is two main memory references are required:

1. To find the frame number
2. To go to the address specified by frame number

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB). Translation Lookaside Buffer (TLB) is a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss), the page number is used as an index while processing the page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.



Steps in TLB hit

1. CPU generates a virtual (logical) address.
2. It is checked in TLB (present).
3. The corresponding frame number is retrieved, which now tells where the main memory page lies.

Steps in TLB miss

1. CPU generates a virtual (logical) address.
2. It is checked in TLB (not present).
3. Now the page number is matched to the page table residing in the main memory (assuming the page table contains all PTE).
4. The corresponding frame number is retrieved, which now tells where the main memory page lies.
5. The TLB is updated with new PTE (if space is not there, one of the replacement techniques comes into the picture i.e either FIFO, LRU or MFU etc).

5.37 What is Next Fit?

The next fit is a modified version of '[first fit](#)'. It begins as the first fit to find a free partition but when called next time it starts searching from where it left off, not from the beginning. This policy makes use of a roving pointer. The pointer moves along the memory chain to search for the next fit. This helps in, to avoid the usage of memory always from the head (beginning) of the free blockchain.

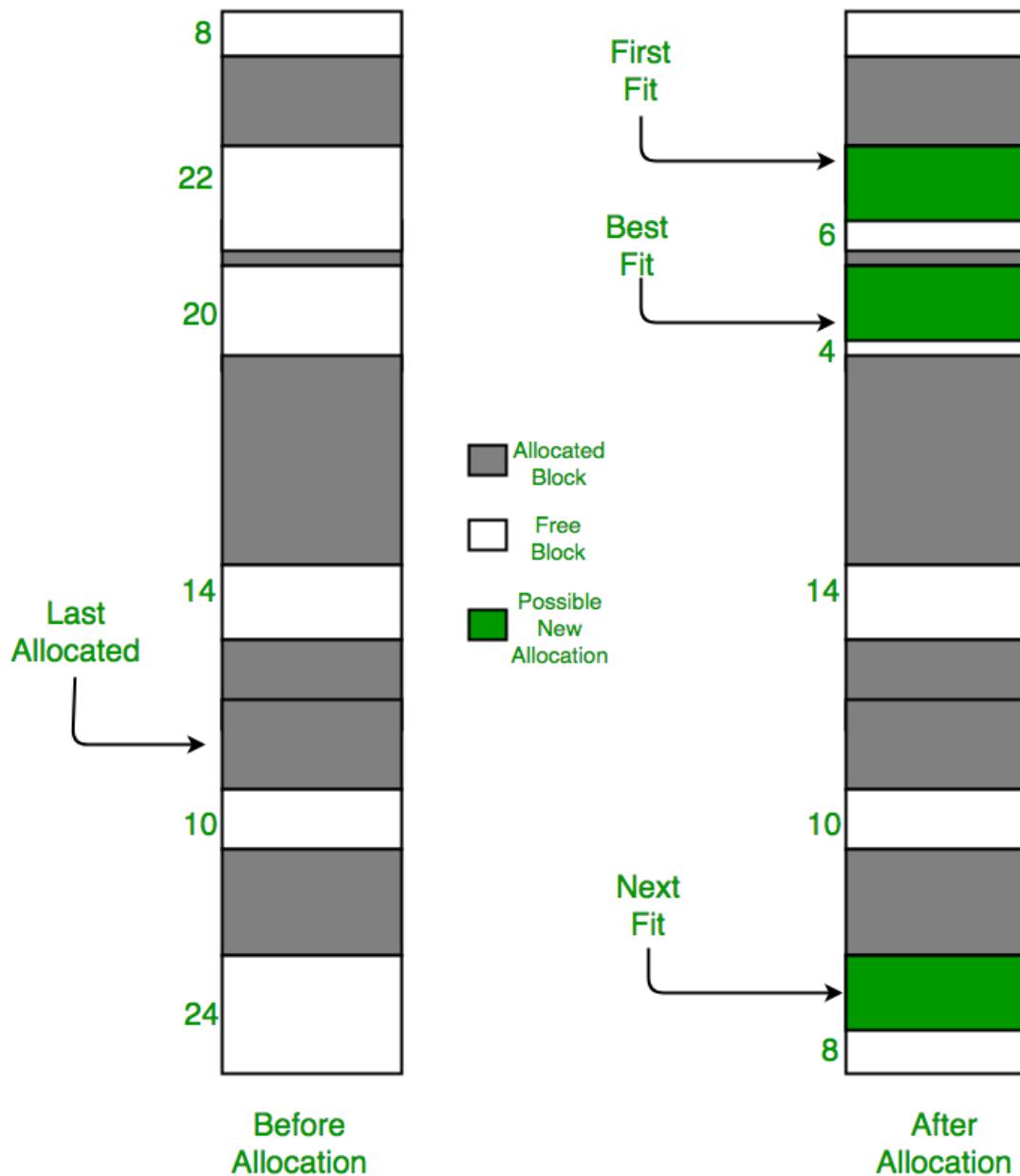
5.38 What is its advantage over first fit?

- First fit is a straight and fast algorithm, but tends to cut a large portion of free parts into small pieces due to which, processes that need a large portion of memory block would not get anything even if the sum of all small pieces is greater than it required which is so-called external fragmentation problem.
- Another problem of the first fit is that it tends to allocate memory parts at the beginning of the memory, which may lead to more internal fragments at the beginning. Next fit tries to address this problem by starting the search for the free portion of parts not from the start of the memory, but from where it ends last time.
- Next fit is a very fast searching algorithm and is also comparatively faster than First Fit and Best Fit Memory Management Algorithms.

Algorithm:

1. Input the number of memory blocks and their sizes and initializes all the blocks as free.
2. Input the number of processes and their sizes.
3. Start by picking each process and check if it can be assigned to the current block, if yes, allocate the required memory and check for next process but from the block where we left not from starting.
4. If the current block size is smaller then keep checking the further blocks.

Memory Allocation before and after allocation of 16 M of memory



In memory management, overlays refer to a technique used to manage memory efficiently by overlaying a portion of memory with another program or data.

The idea behind overlays is to only load the necessary parts of a program into memory at a given time, freeing up memory for other tasks. The unused portions of the program are kept on disk or other storage, and are loaded into memory as needed. This allows programs to be larger than the available memory, but still run smoothly.

The main problem in Fixed partitioning is the size of a process has to be limited by the maximum size of the partition, which means a process can never be span over another. In order to solve this problem, earlier people have used some solution which is called as Overlays.

The concept of **overlays** is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it and once the part is done, then you just unload it, means just pull it back and get the new part you required and run it.

Formally,

"The process of **transferring a block** of program code or other data into internal memory, replacing what is already stored".

Sometimes it happens that compare to the size of the biggest partition, the size of the program will be even more, then, in that case, you should go with overlays.

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

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

Page Replacement Algorithms:

1. First In First Out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find the number of page faults.

Page reference	1, 3, 0, 3, 5, 6, 3
1	3
0	0
3	0
5	0
6	0
3	3
1	1
1	1
1	1
5	5
5	5
5	5

Miss Miss Miss Hit Miss Miss Miss

Total Page Fault = 6

Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.

when 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault. 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault. Finally, when 3 come it is not available so it replaces 0 1 page fault.

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

2. Optimal Page replacement: In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.

Page reference	7,0,1,2,0,3,0,4,2,3,0,3,2,3	No. of Page frame - 4
7	0	1
0	2	2
1	1	1
0	0	0
3	2	1
0	2	1
4	2	4
2	2	4
3	2	4
0	2	4
3	2	4
2	2	0
3	3	0
7	7	3
7	7	3
Miss	Miss	Miss
Miss	Miss	Hit
Miss	Miss	Miss
Hit	Hit	Hit
Miss	Miss	Hit
Hit	Hit	Hit
Hit	Hit	Hit
Hit	Hit	Hit
Total Page Fault = 6		
Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots → 4 Page faults		
0 is already there so → 0 Page fault . when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. → 1 Page fault . 0 is already there so → 0 Page fault . 4 will takes place of 1 → 1 Page Fault .		
Now for the further page reference string → 0 Page fault because they are already available in the memory.		
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.		
3. Least Recently Used: In this algorithm, page will be replaced which is least recently used.		
Example-3: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.		

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

0 is already there so \rightarrow **0 Page fault**. when 3 came it will take the place of 7 because it is least recently used \rightarrow **1 Page fault**

0 is already in memory so → **0 Page fault.**

4 will takes place of 1 \rightarrow 1 Page Fault

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

4. Most Recently Used (MRU): In this algorithm, page will be replaced which has been used recently. Belady's anomaly can occur in this algorithm.

Page reference	7,0,1,2,0,3,0,4,2,3,0,3,2,3	No. of Page frame - 4
7	0	1
0	1	2
7	0	1
7	0	2
7	1	1
7	2	0
7	3	3
7	0	1
4	2	2
7	1	1
7	4	3
7	4	1
7	4	4
7	7	1
7	7	2
7	7	3
Miss	Miss	Miss
Miss	Hit	Miss
Miss	Miss	Miss
Miss	Miss	Miss
Total Page Fault = 12		

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

0 is already there so—> **0 page fault**

when 3 comes it will take place of 0 because it is most recently used —>**1 Page fault**

when 0 comes it will take place of 3 —>**1 Page fault**

when 4 comes it will take place of 0 —>**1 Page fault**

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

when 3 comes it will take place of 2 —>**1 Page fault**

when 0 comes it will take place of 3 —>**1 Page fault**

when 3 comes it will take place of 0 —>**1 Page fault**

when 2 comes it will take place of 3 —>**1 Page fault**

when 3 comes it will take place of 2 —>**1 Page fault**

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In **Least Recently Used (LRU)** algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.

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

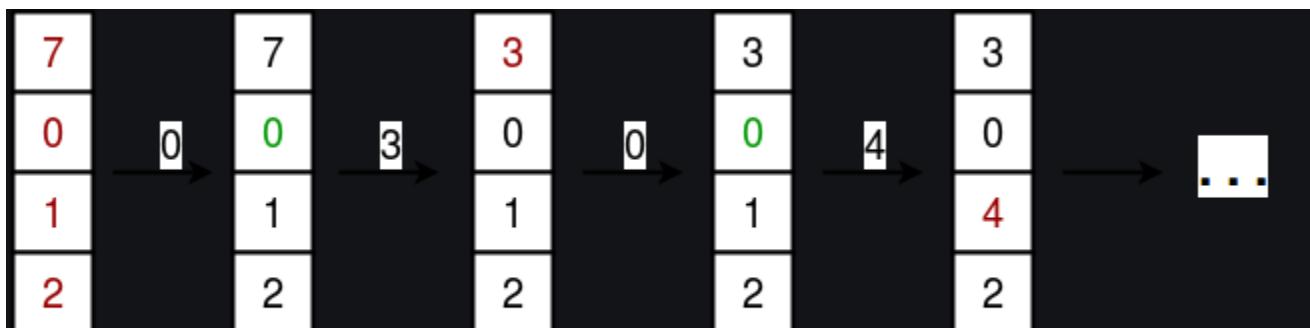
0 is already there so —> **0 Page fault.**

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

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

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

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



Total Page faults = 6

Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.

Below is implementation of above steps.

- Java

```
// Java implementation of above algorithm

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

class Test
{
    // Method to find page faults using indexes
    static int pageFaults(int pages[], int n, int capacity)
    {
        // To represent set of current pages. We use
        // an unordered_set so that we quickly check
        // if a page is present in set or not
        HashSet<Integer> s = new HashSet<>(capacity);

        // To store least recently used indexes
        // of pages.
        HashMap<Integer, Integer> indexes = new HashMap<>();

        // Start from initial page
        int page_faults = 0;
        for (int i=0; i<n; i++)
        {
            // Check if the set can hold more pages
            if (s.size() < capacity)
            {
                // Insert it into set if not present
                // already which represents page fault
                if (!s.contains(pages[i]))
                {
                    s.add(pages[i]);

                    // increment page fault
                    page_faults++;
                }
            }

            // Store the recently used index of
            // each page
            indexes.put(pages[i], i);
        }

        // If the set is full then need to perform lru
    }
}
```

```

        // i.e. remove the least recently used page
        // and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (!s.contains(pages[i]))
        {
            // Find the least recently used pages
            // that is present in the set
            int lru = Integer.MAX_VALUE, val=Integer.MIN_VALUE;

            Iterator<Integer> itr = s.iterator();

            while (itr.hasNext()) {
                int temp = itr.next();
                if (indexes.get(temp) < lru)
                {
                    lru = indexes.get(temp);
                    val = temp;
                }
            }

            // Remove the indexes page
            s.remove(val);
            //remove lru from hashmap
            indexes.remove(val);
            // insert the current page
            s.add(pages[i]);

            // Increment page faults
            page_faults++;
        }

        // Update the current page index
        indexes.put(pages[i], i);
    }
}

return page_faults;
}

// Driver method
public static void main(String args[])

```

```

{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

    int capacity = 4;

    System.out.println(pageFaults(pages, pages.length, capacity));
}
}

// This code is contributed by Gaurav Miglani

```

Output:

Complexity Analysis :

- **Time Complexity** : average time complexity of set and map operations is O(1) and the worst-case time complexity is O(n) but O(n) is the dominant term.
- **Space Complexity** : O(capacity) which is a constant and depends on the size of the input array and the size of the memory buffer.

```

// Java program for page replacement algorithms
import java.util.ArrayList;

public class LRU {

    // Driver method
    public static void main(String[] args) {
        int capacity = 4;
        int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

        // To represent set of current pages.We use
        // an ArrayList
        ArrayList<Integer> s=new ArrayList<>(capacity);
        int count=0;
        int page_faults=0;
        for(int i:arr)
        {
            // Insert it into set if not present
            // already which represents page fault
            if(!s.contains(i))
            {

                // Check if the set can hold equal pages
                if(s.size()==capacity)
                {

```

```

        s.remove(0);
        s.add(capacity-1,i);
    }
    else
        s.add(count,i);
        // Increment page faults
        page_faults++;
        ++count;

    }
    else
    {
        // Remove the indexes page
        s.remove((Object)i);
        // insert the current page
        s.add(s.size(),i);
    }

}
System.out.println(page_faults);
}
}

```

Least Frequently Used (LFU) is a caching algorithm in which the least frequently used cache block is removed whenever the cache is overflowed. In LFU we check the old page as well as the frequency of that page and if the frequency of the page is larger than the old page we cannot remove it and if all the old pages are having same frequency then take last i.e FIFO method for that and remove that page.

Min-heap data structure is a good option to implement this algorithm, as it handles insertion, deletion, and update in logarithmic time complexity. A tie can be resolved by removing the least recently used cache block. The following two containers have been used to solve the problem:

- A vector of integer pairs has been used to represent the cache, where each pair consists of the block number and the number of times it has been used. The vector is ordered in the form of a min-heap, which allows us to access the least frequently used block in constant time.
- A hashmap has been used to store the indices of the cache blocks which allows searching in constant time.

Below is the implementation of the above approach:

- Java

```
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

class Pair {
    int value, frequency;

    public Pair(int value, int frequency)
    {
        this.value = value;
        this.frequency = frequency;
    }
}

class LFU {
    int cacheSize;
    Map<Integer, Pair> cache;

    public LFU(int cacheSize)
    {
        this.cacheSize = cacheSize;
        this.cache = new HashMap<Integer, Pair>();
    }

    // Self made heap to Rearranges
    // the nodes in order to maintain the heap property
    public void increment(int value)
    {
        if (cache.containsKey(value)) {
            cache.get(value).frequency += 1;
        }
    }

    // Function to Insert a new node in the heap
    public void insert(int value)
    {
        if (cache.size() == cacheSize) {
            // remove least frequently used
            int lfuKey = findLFU();
            System.out.println("Cache block " + lfuKey
                + " removed.");
            cache.remove(lfuKey);
        }
    }
}
```

```
        Pair newPair = new Pair(value, 1);
        cache.put(value, newPair);
        System.out.println("Cache block " + value
                           + " inserted.");
    }

    // Function to refer to the block value in the cache
    public void refer(int value)
    {
        if (!cache.containsKey(value)) {
            insert(value);
        }
        else {
            increment(value);
        }
    }

    // Function to find the LFU block
    public int findLFU()
    {
        int lfuKey = 0;
        int minFrequency = Integer.MAX_VALUE;
        for (Map.Entry<Integer, Pair> entry :
             cache.entrySet()) {
            if (entry.getValue().frequency < minFrequency) {
                minFrequency = entry.getValue().frequency;
                lfuKey = entry.getKey();
            }
        }
        return lfuKey;
    }
}

public class Main {
    public static void main(String[] args)
    {
        LFU lfuCache = new LFU(4);
        lfuCache.refer(1);
        lfuCache.refer(2);
        lfuCache.refer(1);
        lfuCache.refer(3);
        lfuCache.refer(2);
        lfuCache.refer(4);
    }
}
```

```
        lfuCache.refer(5);
    }
}
```

Thrashing is a condition or a situation when the system is spending a major portion of its time servicing the page faults, but the actual processing done is very negligible.

Causes of thrashing:

1. High degree of multiprogramming.
2. Lack of frames.
3. Page replacement policy.

Thrashing's Causes

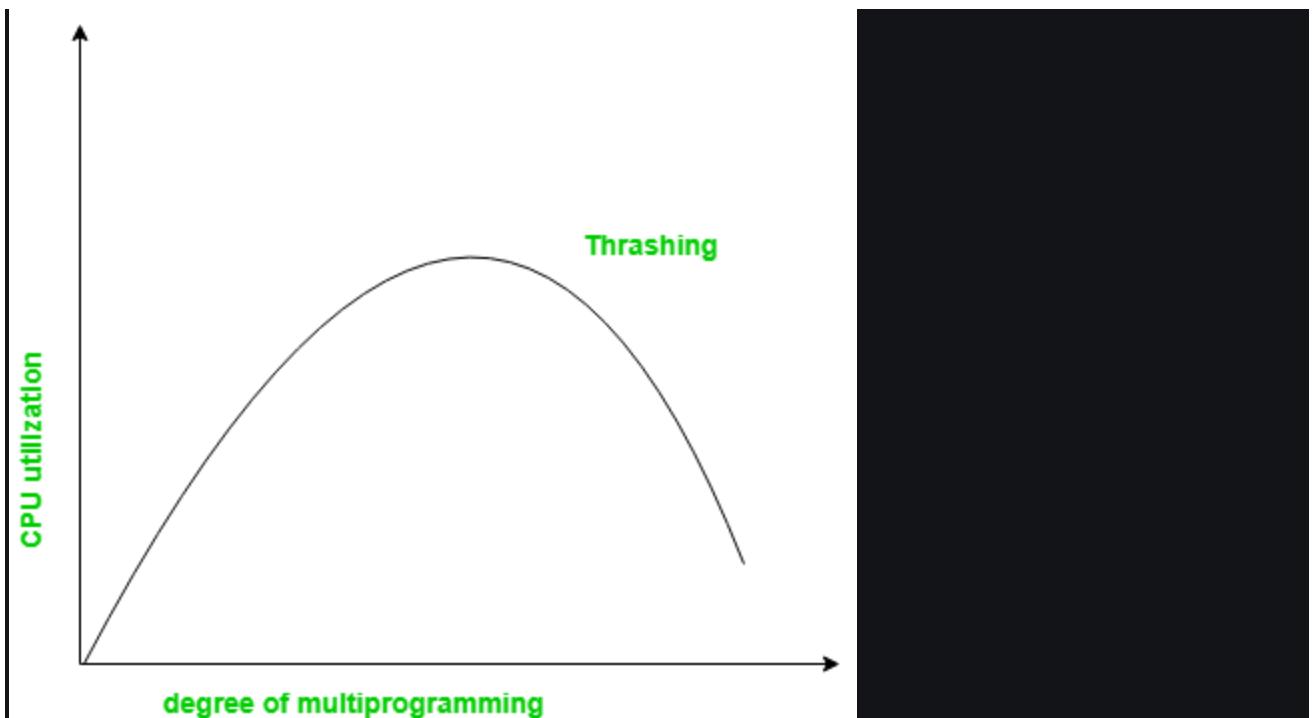
Thrashing has an impact on the operating system's execution performance. Thrashing also causes serious performance issues with the operating system. When the CPU's usage is low, the process scheduling mechanism tries to load multiple processes into memory at the same time, increasing the degree of Multi programming.

In this case, the number of processes in the memory exceeds the number of frames available in the memory. Each process is given a set number of frames to work with.

If a high-priority process arrives in memory and the frame is not vacant at the moment, the other process occupying the frame will be moved to secondary storage, and the free frame will be allotted to a higher-priority process.

We may also argue that as soon as the memory is full, the procedure begins to take a long time to swap in the required pages. Because most of the processes are waiting for pages, the CPU utilization drops again.

As a result, a high level of multi programming and a lack of frames are two of the most common reasons for thrashing in the operating system.



The basic concept involved is that if a process is allocated to few frames, then there will be too many and too frequent page faults. As a result, no useful work would be done by the CPU and the CPU utilization would fall drastically. The long-term scheduler would then try to improve the CPU utilization by loading some more processes into the memory thereby increasing the degree of multi programming. This would result in a further decrease in the CPU utilization triggering a chained reaction of higher page faults followed by an increase in the degree of multi programming, called Thrashing.

Locality Model –

A locality is a set of pages that are actively used together. The locality model states that as a process executes, it moves from one locality to another. A program is generally composed of several different localities which may overlap.

For example, when a function is called, it defines a new locality where memory references are made to the instructions of the function call, its local and global variables, etc. Similarly, when the function is exited, the process leaves this locality.

Techniques to handle:

1. Working Set Model –

This model is based on the above-stated concept of the Locality Model.

The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

The accuracy of the working set is dependent on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

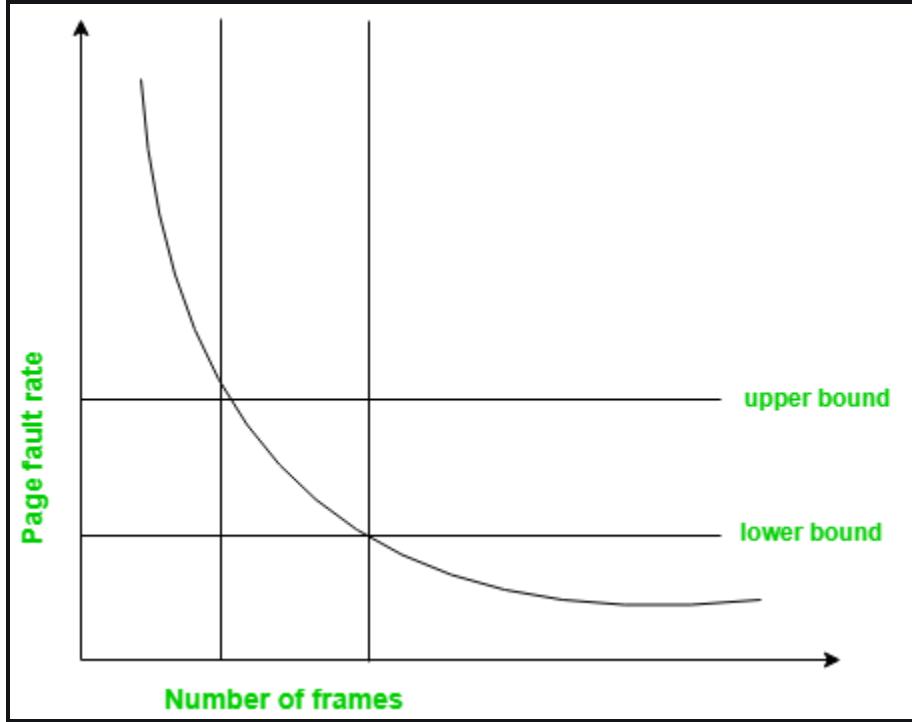
If D is the total demand for frames and w_i is the working set size for process i,

Now, if 'm' is the number of frames available in the memory, there are 2 possibilities:

- (i) $D > m$ i.e. total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
- (ii) $D \leq m$, then there would be no thrashing.

2. Page Fault Frequency –

A more direct approach to handling thrashing is the one that uses the Page-Fault Frequency concept.



The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate.

If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.

Upper and lower limits can be established on the desired page fault rate as shown in the

diagram.

If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page fault rate exceeds the upper limit, more frames can be allocated to the process.

In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagram.

Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocated to them can be reallocated to other processes. The suspended processes can then be restarted later.

Allocating kernel memory is a critical task in operating system design, as the kernel needs to manage memory efficiently and effectively to ensure optimal system performance. Two common methods for allocating kernel memory are the buddy system and the slab system.

The buddy system is a memory allocation algorithm that works by dividing memory into blocks of a fixed size, with each block being a power of two in size. When a request for memory is made, the algorithm finds the smallest available block of memory that is large enough to satisfy the request. If the block is larger than the requested size, it is split into two smaller blocks of equal size (the “buddies”), with each buddy being marked as either allocated or free. The algorithm then continues recursively until it finds the exact size of the requested memory or a block that is the smallest possible size.

The slab system is a memory allocation algorithm that is designed specifically for kernel memory. It works by dividing memory into fixed-size caches or slabs, each of which contains a set of objects of the same type. When a request for memory is made, the algorithm first checks if there is an available object in the appropriate slab cache. If there is, the object is returned. If not, the algorithm allocates a new slab and adds it to the appropriate cache.

The advantages of the buddy system are that it is easy to implement and can handle a wide range of memory sizes. The disadvantages are that it can lead to memory fragmentation and that it can be inefficient for allocating small amounts of memory.

The advantages of the slab system are that it is efficient for allocating small amounts of memory and that it can prevent memory fragmentation. The disadvantages are that it can be more complex to implement than the buddy system and that it may require more memory overhead.

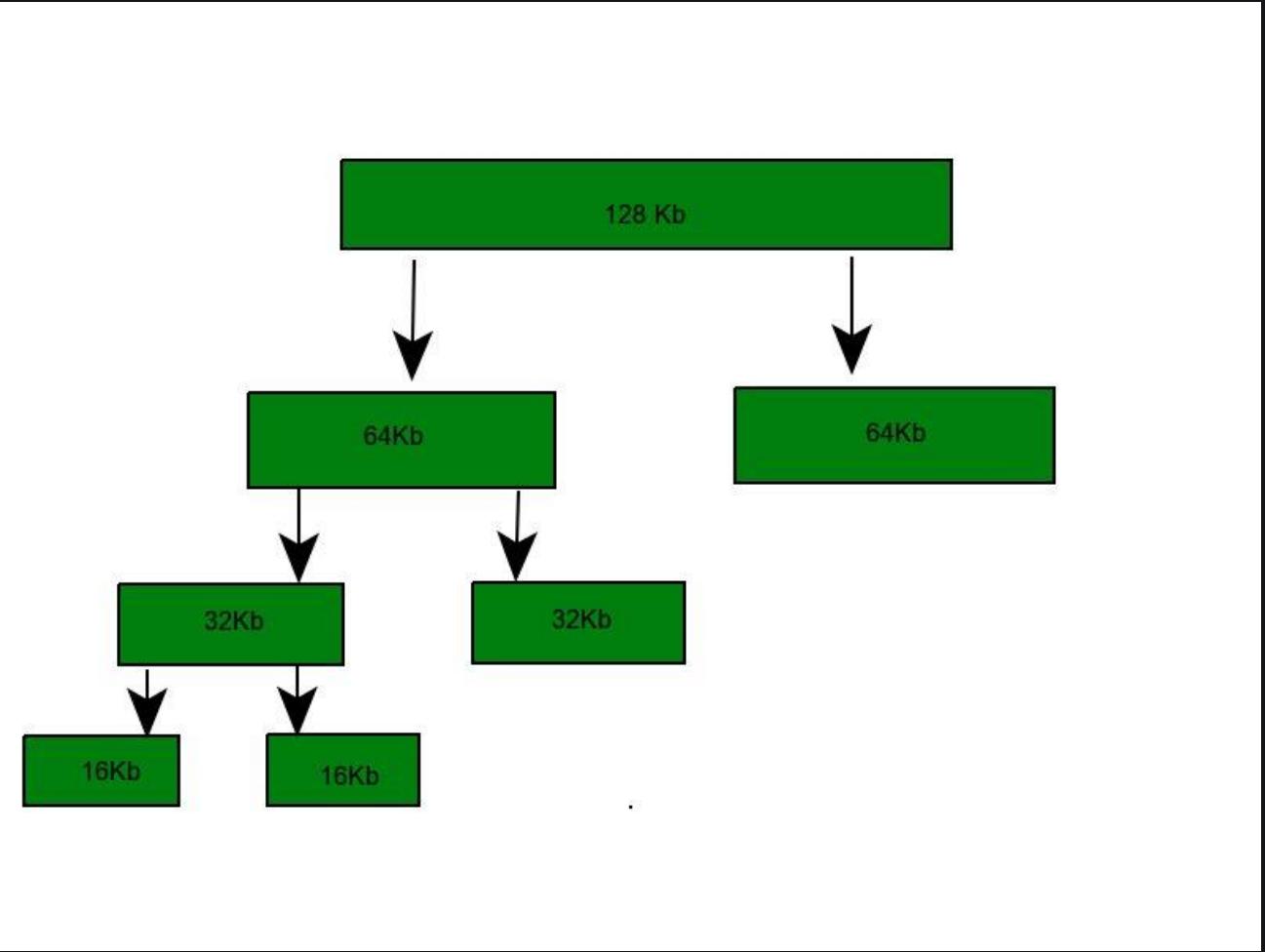
Overall, both the buddy system and the slab system are effective methods for allocating kernel memory, and the choice between the two depends on the specific needs and requirements of the operating system being developed.

Two strategies for managing free memory that is assigned to kernel processes:

1. Buddy system –

Buddy allocation system is an algorithm in which a larger memory block is divided into small parts to satisfy the request. This algorithm is used to give best fit. The two smaller parts of block are of equal size and called as buddies. In the same manner one of the two buddies will further divide into smaller parts until the request is fulfilled. Benefit of this technique is that the two buddies can combine to form the block of larger size according to the memory request.

Example – If the request of 25Kb is made then block of size 32Kb is allocated.



Four Types of Buddy System –

1. Binary buddy system
2. Fibonacci buddy system
3. Weighted buddy system
4. Tertiary buddy system

Why buddy system?

If the partition size and process size are different then poor match occurs and may use space inefficiently.

It is easy to implement and efficient than dynamic allocation.

Binary buddy system –

The buddy system maintains a list of the free blocks of each size (called a free list), so that

it is easy to find a block of the desired size, if one is available. If no block of the requested size is available, Allocate searches for the first non-empty list for blocks of atleast the size requested. In either case, a block is removed from the free list.

Example – Assume the size of memory segment is initially 256kb and the kernel requests 25kb of memory. The segment is initially divided into two buddies. Let we call A1 and A2 each 128kb in size. One of these buddies is further divided into two 64kb buddies let say B1 and B2. But the next highest power of 25kb is 32kb so, either B1 or B2 is further divided into two 32kb buddies(C1 and C2) and finally one of these buddies is used to satisfy the 25kb request. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

Fibonacci buddy system –

This is the system in which blocks are divided into sizes which are Fibonacci numbers. It satisfies the following relation:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 144, 233, 377, 610. The address calculation for the binary and weighted buddy systems is straight forward, but the original procedure for the Fibonacci buddy system was either limited to a small, fixed number of block sizes or a time consuming computation.

Advantages –

- In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation.
- The buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.
- The buddy system is very fast to allocate or deallocate memory.
- In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit algorithms.
- Other advantage is coalescing.
- Address calculation is easy.
- Efficient memory management: Kernel memory allocators are designed to efficiently manage memory resources, ensuring that memory is allocated and deallocated in a way that minimizes fragmentation and maximizes available space.
- Customizable allocation policies: Kernel memory allocators can be customized to implement specific allocation policies, such as best fit, worst fit, or first fit. This allows for more precise control over memory usage and allocation.
- High performance: Kernel memory allocators are typically designed for high-performance systems, allowing for fast and efficient allocation and deallocation of memory.
- Reduced memory overhead: Memory allocators can reduce memory overhead by allowing for more efficient use of memory resources, reducing the overall memory footprint of the system.

- Improved system stability: Efficient memory management can help improve system stability by reducing the likelihood of memory leaks, crashes, and other issues related to memory allocation.

Overall, using a kernel memory allocator can help improve system performance, stability, and memory management, making it an essential component of any operating system.

What is coalescing?

It is defined as how quickly adjacent buddies can be combined to form larger segments this is known as coalescing.

For example, when the kernel releases the C1 unit it was allocated, the system can coalesce C1 and C2 into a 64kb segment. This segment B1 can in turn be coalesced with its buddy B2 to form a 128kb segment. Ultimately we can end up with the original 256kb segment.

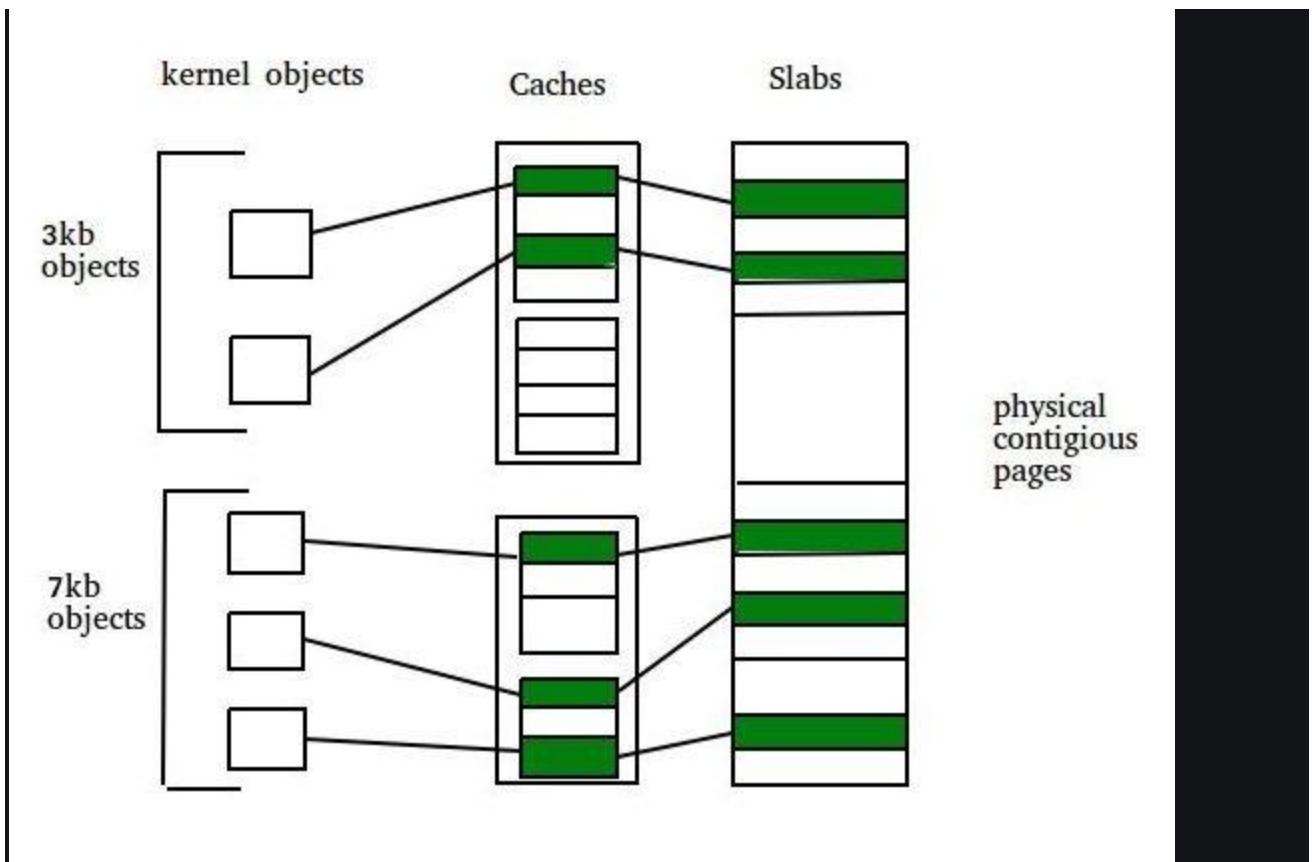
Drawback –

The main drawback in buddy system is internal fragmentation as larger block of memory is acquired then required. For example if a 36 kb request is made then it can only be satisfied by 64 kb segment and remaining memory is wasted.

2. Slab Allocation –

A second strategy for allocating kernel memory is known as slab allocation. It eliminates fragmentation caused by allocations and deallocations. This method is used to retain allocated memory that contains a data object of a certain type for reuse upon subsequent allocations of objects of the same type. In slab allocation memory chunks suitable to fit data objects of certain type or size are preallocated. Cache does not free the space immediately after use although it keeps track of data which are required frequently so that whenever request is made the data will reach very fast. Two terms required are:

- **Slab** – A slab is made up of one or more physically contiguous pages. The slab is the actual container of data associated with objects of the specific kind of the containing cache.
- **Cache** – Cache represents a small amount of very fast memory. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure.



Example –

- A separate cache for a data structure representing processes descriptors
- Separate cache for file objects
- Separate cache for semaphores etc.

Each cache is populated with objects that are instantiations of the kernel data structure the cache represents. For example the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects.

Implementation –

The slab allocation algorithm uses caches to store kernel objects. When a cache is created a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on size of the associated slab.

Example – A 12 kb slab (made up of three contiguous 4 kb pages) could store six 2 kb objects. Initially all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

In linux, a slab may in one of three possible states:

1. **Full** – All objects in the slab are marked as used
2. **Empty** – All objects in the slab are marked as free
3. **Partial** – The slab consists of both

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache.

Benefits of slab allocator –

- No memory is wasted due to fragmentation because each unique kernel data structure has an associated cache.
- Memory request can be satisfied quickly.
- The slab allocating scheme is particularly effective for managing when objects are frequently allocated or deallocated. The act of allocating and releasing memory can be a time consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. When the kernel has finished with an object and releases it, it is marked as free and return to its cache, thus making it immediately available for subsequent request from the kernel.
- When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program.
- A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.
- **Static Linking and Static Libraries**
- is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are,
- **.a**
- files in Linux and
- **.lib**
- files in Windows.
- **Steps to create a static library**

Let us create and use a Static Library in UNIX or UNIX like OS.

1.

Create a C file that contains functions in your library.

- C

```
/* Filename: lib_mylib.c */
#include <stdio.h>
void fun(void)
{
    printf("fun() called from a static library");
}
```

We have created only one file for simplicity. We can also create multiple files in a library.

2.

Create a header file for the library

```
/* Filename: lib_mylib.h */  
void fun(void);
```

3.

Compile library files.

4.

Create static library. This step is to bundle multiple object files in one static library (see [ar](#) for details). The output of this step is static library.

5.

Now our static library is ready to use. At this point we could just copy lib_mylib.a somewhere else to use it. For demo purposes, let us keep the library in the current directory.

Let us create a driver program that uses above created static library

.

1.

Create a C file with main function

```
• C
```

```
/* filename: driver.c */  
#include "lib_mylib.h"  
void main()  
{  
    fun();  
}
```

2.

Compile the driver program.

3.

Link the compiled driver program to the static library. Note that -L

.

is used to tell that the static library is in current folder (See [this](#)

for details of -L and -l options).

4.

Run the driver program

Following are some important points about static libraries.

1.

For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.

2.

Each process gets its own copy of the code and data. Whereas in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.

3.

Since library code is connected at compile time, the final executable has no dependencies on the library at run time i.e. no additional run-time loading costs, it means that you don't need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.

4.

In static libraries, once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system.

5.

One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.

6.

One major advantage of static libraries being preferred even now "is speed". There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

Dynamic linking and Dynamic Libraries

Dynamic Linking doesn't require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are,

.so

in Linux and

.dll

in Windows. We will soon be covering more points on Dynamic Libraries and steps to create them. This article is compiled by

Shared libraries are useful in sharing code which is common across many applications. For example, it is more economic to pack all the code related to TCP/IP implementation in a shared library. However, data can't be shared as every application needs its own set of

data. Applications like, browser, ftp, telnet, etc... make use of the shared ‘network’ library to elevate specific functionality.

Every operating system has its own representation and tool-set to create shared libraries. More or less the concepts are same. On Windows every object file (*.obj, *.dll, *.ocx, *.sys, *.exe etc...) follow a format called Portable Executable. Even shared libraries (called as Dynamic Linked Libraries or DLL in short) are also represented in PE format. The tool-set that is used to create these libraries need to understand the binary format. Linux variants follow a format called Executable and Linkable Format (ELF). The ELF files are position independent (PIC) format. Shared libraries in Linux are referred as shared objects (generally with extension *.so). These are similar to DLLs in Windows platform. Even shared object files follow the ELF binary format.

Remember, the file extensions (*.dll, *.so, *.a, *.lib, etc...) are just for programmer convenience. They don’t have any significance. All these are binary files. You can name them as you wish. Yet ensure you provide absolute paths in building applications.

In general, when we compile an application the steps are simple. Compile, Link and Load. However, it is not simple. These steps are more versatile on modern operating systems.

When you link your application against static library, the code is part of your application. There is no dependency. Even though it causes the application size to increase, it has its own advantages. The primary one is speed as there will be no symbol (a program entity) resolution at runtime. Since every piece of code part of the binary image, such applications are independent of version mismatch issues. However, the cost is on fixing an issue in library code. If there is any bug in library code, entire application need to be recompiled and shipped to the client. In case of dynamic libraries, fixing or upgrading the libraries is easy. You just need to ship the updated shared libraries. The application need not to recompile, it only need to re-run. You can design a mechanism where we don’t need to restart the application.

When we link an application against a shared library, the linker leaves some stubs (unresolved symbols) to be filled at application loading time. These stubs need to be filled by a tool called, *dynamic linker* at run time or at application loading time. Again loading of a library is of two types, static loading and dynamic loading. Don’t confuse between ***static loading*** vs ***static linking*** and ***dynamic loading*** vs ***dynamic linking***.

For example, you have built an application that depends on *libstdc++.so* which is a shared object (dynamic library). How does the application become aware of required shared libraries? (If you are interested, explore the tools *tdump* from Borland tool set, *objdump* or *nm* or *readelf* tools on Linux).

Static loading:

- In static loading, all of those dependent shared libraries are loaded into memory even before the application starts execution. If loading of any shared library fails, the application won't run.
- A dynamic loader examines application's dependency on shared libraries. If these libraries are already loaded into the memory, the library address space is mapped to application virtual address space (VAS) and the dynamic linker does relocation of unresolved symbols.
- If these libraries are not loaded into memory (perhaps your application might be first to invoke the shared library), the loader searches in standard library paths and loads them into memory, then maps and resolves symbols. Again loading is big process, if you are interested write your own loader :).
- While resolving the symbols, if the dynamic linker not able to find any symbol (maybe due to older version of shared library), the application can't be started.

Dynamic Loading:

- As the name indicates, dynamic loading is about loading of library on demand.
- For example, if you want a small functionality from a shared library. Why should it be loaded at the application load time and sit in the memory? You can invoke loading of these shared libraries dynamically when you need their functionality. This is called dynamic loading. In this case, the programmer aware of situation 'when should the library be loaded'. The tool-set and relevant kernel provides API to support dynamic loading, and querying of symbols in the shared library.

[How to create a Dynamic Library](#)

The way to create a Dynamic Library in Linux is with the `gcc` command using the `-c` to generate the object files (.o) from the source files (.c) and the `-fPIC` to make the code position independent. Thus, the following command makes a bunch of .o files from each .c file in the current directory (You can select which functions you want for your library).

```
gcc -c -fPIC *.c
```

Next, we are going to put together those objects files into one library. To do this as a Dynamic Library we also use `gcc` but with the `-shared` option. The `-o` is to specify the name of the file you want it to have.

```
gcc -shared -o liball.so *.o
```

6. Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process:** The execution of one process does not affect the execution of other processes.
- **Cooperative Process:** A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

6.1 Race Condition

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, and then the outcome depends on the particular order in which the access takes place. A [race condition](#) is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Example:

Let's understand one example to understand the race condition better:

Let's say there are two processes P1 and P2 which share a common variable (shared=10), both processes are present in – queue and waiting for their turn to be executed. Suppose, Process P1 first come under execution, and the CPU store a common variable between them (shared=10) in the local variable (X=10) and increment it by 1(X=11), after then when the CPU read line sleep(1),it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in a waiting state for 1 second.

Now CPU execute the Process P2 line by line and store common variable (Shared=10) in its local variable (Y=10) and decrement Y by 1(Y=9), after then when CPU read sleep(1), the current process P2 goes in waiting for state and CPU remains idle for some time as there is no process in ready-queue, after completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code (store the local variable (X=11) in common variable (shared=11)), CPU remain idle for sometime waiting for any process in ready-queue, after completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2(store the local variable (Y=9) in common variable (shared=9)).

Initially Shared = 10

Process 1	Process 2
int X = shared	int Y = shared
X++	Y-
sleep(1)	sleep(1)
shared = X	shared = Y

Note: We are assuming the final value of a common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable (shared=10) by 1 and Process P2 decrement variable (shared=11) by 1 and finally it becomes shared=10). But we are getting undesired value due to a lack of proper synchronization.

Actual meaning of race-condition

- If the order of execution of the process(first P1 -> then P2) then we will get the value of common variable (shared) =9.
- If the order of execution of the process(first P2 -> then P1) then we will get the final value of common variable (shared) =11.
- Here the (value1 = 9) and (value2=10) are racing, If we execute these two processes in our computer system then sometime we will get 9 and sometime we will get 10 as the final value of a common variable(shared). This phenomenon is called race condition.

6.2 Critical Section Problem

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

```

do {

    entry section

    critical section

    exit section

    remainder section

} while (TRUE);

```

In the entry section, the process requests for entry in the **Critical Section**.

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress:** If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

6.3 Peterson's Solution

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.

```

do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j] && turn == j) ;

        critical section

    flag[i] = FALSE ;

        remainder section

    } while (TRUE) ;

```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

6.4 Disadvantages of Peterson's Solution

- It involves busy waiting. (In the Peterson's solution, the code statement- "while(flag[j] && turn == j);;" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

6.5 Semaphores

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that is called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization. A Semaphore is an integer variable, which can be accessed only through two operations

`wait()`

and

`signal()`.

There are two types of semaphores: Binary Semaphores and Counting Semaphores.

- **Binary Semaphores:** They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.
- **Counting Semaphores:** They can have any value and are not restricted to a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

6.6 Advantages of Process Synchronization

- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

6.7 Disadvantages of Process Synchronization

- Adds overhead to the system
- This can lead to performance degradation
- Increases the complexity of the system
- Can cause deadlocks if not implemented properly.

In computer science, a critical section refers to a segment of code that is executed by multiple concurrent threads or processes, and which accesses shared resources. These resources may include shared memory, files, or other system resources that can only be accessed by one thread or process at a time to avoid data inconsistency or race conditions.

1. The critical section must be executed as an atomic operation, which means that once one thread or process has entered the critical section, all other threads or processes must wait until the executing thread or process exits the critical section. The purpose of synchronization mechanisms is to ensure that only one thread or process can execute the critical section at a time.
2. The concept of a critical section is central to synchronization in computer systems, as it is necessary to ensure that multiple threads or processes can execute concurrently without interfering with each other. Various synchronization mechanisms such as semaphores, mutexes, monitors, and

condition variables are used to implement critical sections and ensure that shared resources are accessed in a mutually exclusive manner.

The use of critical sections in synchronization can be advantageous in improving the performance of concurrent systems, as it allows multiple threads or processes to work together without interfering with each other. However, care must be taken in designing and implementing critical sections, as incorrect synchronization can lead to race conditions and deadlocks.

6.8 Critical Section

When more than one [processes](#) try to access the same code segment that segment is known as the critical section. The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.

In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically ([read this post](#) for atomicity), such as accessing a resource (file, input or output port, global data, etc.) In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e., data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.

Few programming languages have built-in support for synchronization. It is critical to understand the importance of race conditions while writing kernel-mode programming (a device driver, kernel thread, etc.) since the programmer can directly access and modify kernel data structures

Although there are some properties that should be followed if any code in the critical section

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
 2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
 3. **Bounded Waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
1. **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
 2. **Non preemptive kernels:** A non preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

The use of critical sections in a program can cause a number of issues, including:

Deadlock: When two or more threads or processes wait for each other to release a critical section, it can result in a deadlock situation in which none of the threads or processes can move. Deadlocks can be difficult to detect and resolve, and they can have a significant impact on a program's performance and reliability.

Starvation: When a thread or process is repeatedly prevented from entering a critical section, it can result in starvation, in which the thread or process is unable to progress. This can happen if the critical section is held for an unusually long period of time, or if a high-priority thread or process is always given priority when entering the critical section.

Overhead: When using critical sections, threads or processes must acquire and release locks or semaphores, which can take time and resources. This may reduce the program's overall performance.

Entry Section

Critical
Section

Exit Section

Remainder
Section

Critical section

It could be visualized using the pseudo-code below –

A simple solution to the critical section can be thought of as shown below,

A thread must acquire a lock prior to executing a critical section. The lock can be acquired by only one thread. There are various ways to implement locks in the above pseudo-code. Let us discuss them in future articles.

Strategies for avoiding problems: While deadlocks, starvation, and overhead are mentioned as potential issues, there are more specific strategies for avoiding or mitigating these problems. For example, using timeouts to prevent deadlocks, implementing priority inheritance to prevent priority inversion and starvation, or optimizing lock implementation to reduce overhead.

Examples of critical sections in real-world applications: While the article describes critical sections in a general sense, it could be useful to provide examples of how critical sections are used in specific real-world applications, such as database management systems or web servers.

Impact on scalability: The use of critical sections can impact the scalability of a program, particularly in distributed systems where multiple nodes are accessing shared resources.

In process synchronization, a critical section is a section of code that accesses shared resources such as variables or data structures, and which must be executed by only one process at a time to avoid race conditions and other synchronization-related issues.

A critical section can be any section of code where shared resources are accessed, and it typically consists of two parts: the entry section and the exit section. The entry section is where a process requests access to the critical section, and the exit section is where it releases the resources and exits the critical section.

To ensure that only one process can execute the critical section at a time, process synchronization mechanisms such as semaphores and mutexes are used. A semaphore is a variable that is used to indicate whether a resource is available or not, while a mutex is a binary semaphore that provides mutual exclusion to shared resources.

When a process enters a critical section, it must first request access to the semaphore or mutex associated with the critical section. If the resource is available, the process can proceed to execute the critical section. If the resource is not available, the process must wait until it is released by the process currently executing the critical section.

Once the process has finished executing the critical section, it releases the semaphore or mutex, allowing another process to enter the critical section if necessary.

Proper use of critical sections and process synchronization mechanisms is essential in concurrent programming to ensure proper synchronization of shared resources and avoid race conditions, deadlocks, and other synchronization-related issues.

Advantages of critical section in process synchronization:

1. **Prevents race conditions:** By ensuring that only one process can execute the critical section at a time, race conditions are prevented, ensuring consistency of shared data.

2. **Provides mutual exclusion:** Critical sections provide mutual exclusion to shared resources, preventing multiple processes from accessing the same resource simultaneously and causing synchronization-related issues.
3. **Reduces CPU utilization:** By allowing processes to wait without wasting CPU cycles, critical sections can reduce CPU utilization, improving overall system efficiency.
4. **Simplifies synchronization:** Critical sections simplify the synchronization of shared resources, as only one process can access the resource at a time, eliminating the need for more complex synchronization mechanisms.

Disadvantages of critical section in process synchronization:

1. **Overhead:** Implementing critical sections using synchronization mechanisms like semaphores and mutexes can introduce additional overhead, slowing down program execution.
2. **Deadlocks:** Poorly implemented critical sections can lead to deadlocks, where multiple processes are waiting indefinitely for each other to release resources.
3. **Can limit parallelism:** If critical sections are too large or are executed frequently, they can limit the degree of parallelism in a program, reducing its overall performance.
4. **Can cause contention:** If multiple processes frequently access the same critical section, contention for the critical section can occur, reducing performance.
5. Overall, critical sections are a useful tool in process synchronization to ensure proper synchronization of shared resources and prevent race conditions. However, they can also introduce additional overhead and can be prone to synchronization-related issues if not implemented correctly.

6.9 Important points related to critical section in process synchronization are:

1. Understanding the concept of critical section and why it's important for synchronization.
2. Familiarity with the different synchronization mechanisms used to implement critical sections, such as semaphores, mutexes, and monitors.
3. Knowledge of common synchronization problems that can arise in critical sections, such as race conditions, deadlocks, and livelocks.
4. Understanding how to design and implement critical sections to ensure proper synchronization of shared resources and prevent synchronization-related issues.
5. Familiarity with best practices for using critical sections in concurrent programming.

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in

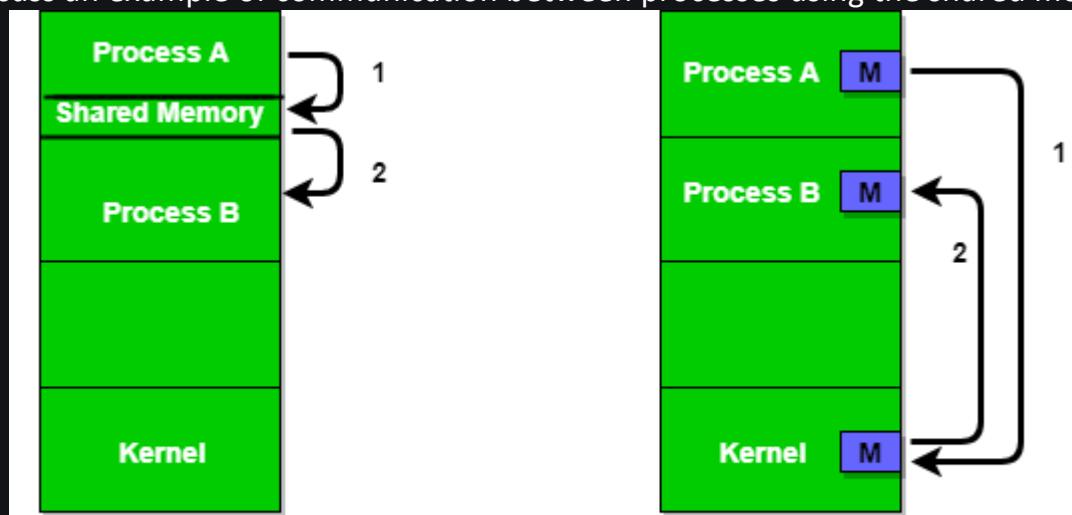
reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Let's discuss an example of communication between processes using the shared memory



method.

Figure 1 - Shared Memory and Message Passing

i)

Shared Memory MethodEx: Producer-Consumer problem

There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or

memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them. The pseudo-code to demonstrate is provided below:

Shared Data between the two Processes

- C

```
#define buff_max 25
#define mod %

struct item{

    // different member of the produced data
    // or consumed data
    -----
}

// An array is needed for holding the items.
// This is the shared place which will be
// access by both process
// item shared_buff [ buff_max ];

// Two variables which will keep track of
// the indexes of the items produced by producer
// and consumer The free index points to
// the next free index. The full index points to
// the first full index.
int free_index = 0;
int full_index = 0;
```

Producer Process Code

- C

```
item nextProduced;

while(1){

    // check if there is no space
    // for production.
    // if so keep waiting.
    while((free_index+1) mod buff_max == full_index);

    shared_buff[free_index] = nextProduced;
    free_index = (free_index + 1) mod buff_max;
}
```

Consumer Process Code

- C

```
item nextConsumed;

while(1){

    // check if there is an available
    // item for consumption.
    // if not keep on waiting for
    // get them produced.
    while((free_index == full_index);

    nextConsumed = shared_buff[full_index];
    full_index = (full_index + 1) mod buff_max;
}
```

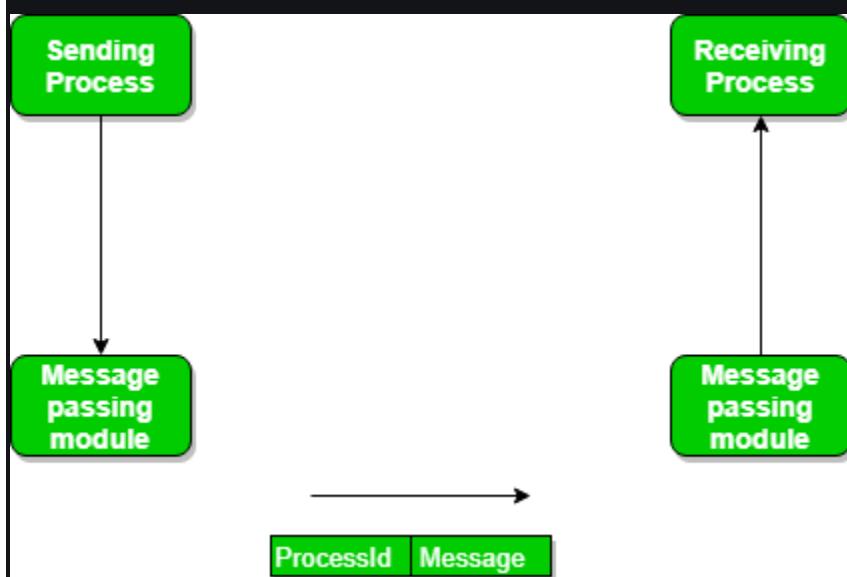
In the above code, the Producer will start producing again when the $(\text{free_index}+1) \bmod \text{buff_max}$ will be free because if it is not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index point to the same index, this implies that there are no items to consume.

Note that the atomic class is used to make sure that the shared variables `free_index` and `full_index` are updated atomically. The mutex is used to protect the critical section where the shared buffer is accessed. The `sleep_for` function is used to simulate the production and consumption of items.

ii) Messaging Passing Method

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

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



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body**.

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

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

like :

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

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

Message Passing through Exchanging the Messages.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver.

Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message

passing. There are basically three preferred combinations:

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

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

e.g. **send(p1, message)** means send the message to p1.

Similarly, **receive(p2, message)** means to receive the message from p2.

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

In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. The standard primitives used are: **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem with this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox that can have multiple senders and a single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver processor when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is created which is shared by n process. The sender is non-blocking and sends the message. The first process which executes the receive will

enter in the critical section and all other processes will be blocking and will wait. Now, let's discuss the Producer-Consumer problem using the message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Producer Code

- C

```
void Producer(void){  
  
    int item;  
    Message m;  
  
    while(1){  
  
        receive(Consumer, &m);  
        item = produce();  
        build_message(&m , item ) ;  
        send(Consumer, &m);  
    }  
}
```

Consumer Code

- C

```
void Consumer(void){  
  
    int item;  
    Message m;  
  
    while(1){  
  
        receive(Producer, &m);  
        item = extracted_item();  
        send(Producer, &m);  
        consume_item(item);  
    }  
}
```

Examples of IPC systems

1. Posix : uses shared memory method.

2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

Communication in client/server Architecture:

There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

Inter Process Communication through shared memory is a concept where two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

6.10 Used System Calls

The system calls that are used in the program are:

Function	Signature	Description
ftok()	key_t ftok()	It is used to generate a unique key.
shmget()	int shmget(key_t key, size_t size, int shmflg);	Upon successful completion, shmget() returns an identifier for the shared memory segment.
shmat()	void *shmat(int shmid, void *shmaddr, int shmflg);	Before you can use a shared memory segment, you have to attach yourself to it using shmat(). Here, shmid is a

Function	Signature	Description
		shared memory ID and shmaddr specifies the specific address to use but we should set it to zero and OS will automatically choose the address.
shmdt()	int shmdt(void *shmaddr);	When you're done with the shared memory segment, your program should detach itself from it using shmdt().
shmctl()	shmctl(int shmid,IPC_RMID,NULL);	When you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

6.11 Shared Memory for Writer Process

6.12 CPP

```
#include <iostream>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile", 65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // shmat to attach to shared memory
    char* str = (char*)shmat(shmid, (void*)0, 0);

    cout << "Write Data : ";
    cin.getline(str, 1024);

    cout << "Data written in memory: " << str << endl;
```

```

        // detach from shared memory
        shmdt(str);

    return 0;
}

```

6.13 Shared Memory for Reader Process

6.14 CPP

```

#include <iostream>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile", 65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // shmat to attach to shared memory
    char* str = (char*)shmat(shmid, (void*)0, 0);

    cout << "Data read from memory:" << str;

    // detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}

```

Output

The screenshot shows two terminal windows side-by-side. Both windows have a dark background and light-colored text. The left window shows the output of the writer process, and the right window shows the output of the reader process.

Left Terminal (Writer Process):

```

andres@andres: ~/Programs/OS$ ./writer
Write Data : Geeks for Geeks
Data written in memory: Geeks for Geeks
andres@andres:~/Programs/OS$ 

```

Right Terminal (Reader Process):

```

andres@andres: ~/Programs/OS$ ./reader
Data read from memory: Geeks for Geeks
andres@andres:~/Programs/OS$ 

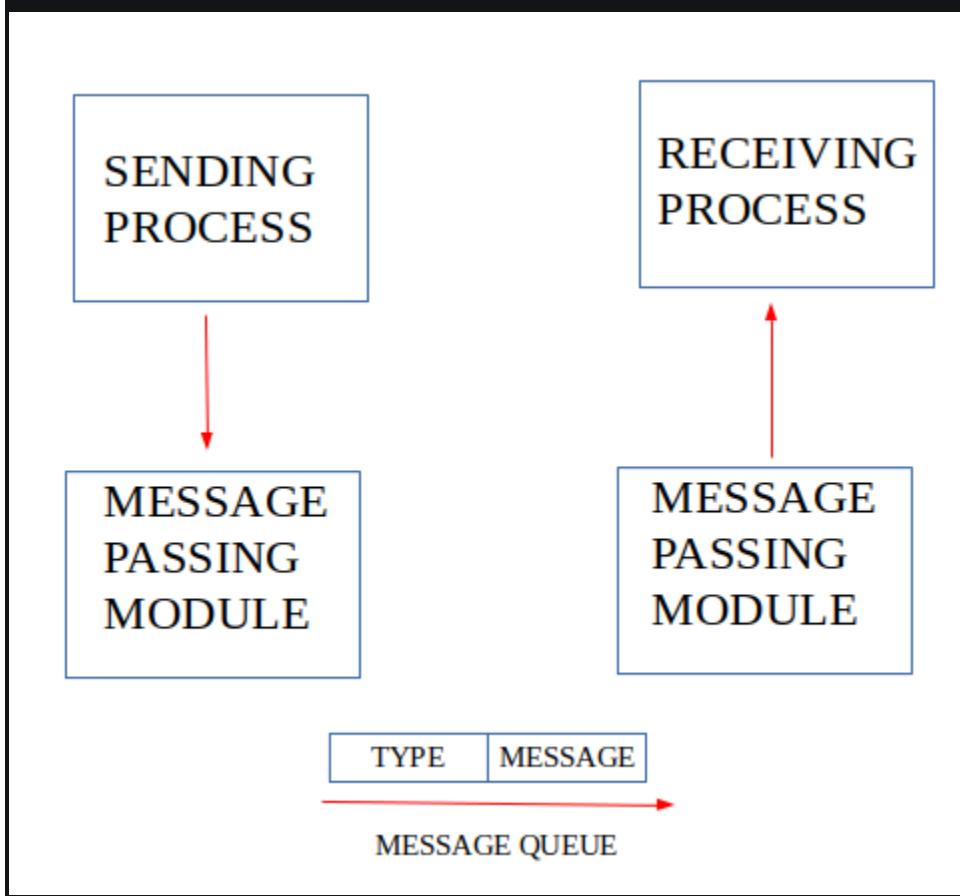
```

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened

by **msgget()**.

New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.



System calls used for message queues:

- **ftok()**: is use to generate a unique key.
- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd().

- **msgrcv()**: messages are retrieved from a queue.
- **msgctl()**: It performs various operations on a queue. Generally it is used to destroy message queue.

MESSAGE QUEUE FOR WRITER PROCESS

- C

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    fgets(message.mesg_text,MAX,stdin);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
}
```

MESSAGE QUEUE FOR READER PROCESS

- C

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
           message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Output:

The image shows two terminal windows side-by-side. Both windows have a title bar 'andres@andres: ~/Programs/OS' and a menu bar 'File Edit View Search Terminal Help'. The left terminal window contains the command 'andres@andres:~/Programs/OS\$./writer' followed by the output 'Write Data : Geeks for Geeks' and 'Data send is : Geeks for Geeks'. The right terminal window contains the command 'andres@andres:~/Programs/OS\$./reader' followed by the output 'Data Received is : Geeks for Geeks'. This illustrates a simple message passing between two programs.

- **Message Passing** – In this model, the concept of message as the major abstraction of model is introduced. The units which inter-change the data and information that is explicitly encode, in the form of message. According to this model, the schema and content of message changes or varies. Message Passing Interface and OpenMP are major examples of this type of model.
- **Remote Procedure Call** – This model explores the keys of procedure call beyond the restrictions of a single process, thus pointing the execution of program in remote processes. In this, primary client-server is implied. A remote process maintains a server component, thus enabling client processes to invoke the approaches and returns the output of the execution. Messages, created by the Remote Procedure Call (RPC) implementation, retrieve the information of the procedure itself and that procedure is to execute having necessary arguments and also returns the values. The use of messages regarding this referred as marshaling of the arguments and return values.
- **Distributed Objects** – It is an implementation of Remote Procedure Call (RPC) model for the Object-oriented model, and contextualizes this for the remote invocation of methods extended by objects. Every process assigns a set of interfaces which are accessible remotely. Client process can demand a reference to these interfaces and invoke the methods available through them. The basic runtime infrastructure is in transformation the local method calls into a request to a remote process and collecting the result of the execution. The interaction within the caller and the remote process is done through messages. This model is stateless by design, distributed object models introduce the complexity of object state management and lifetime. Common Object Request Broker Architecture (CORBA), Component Object Model (COM, DCOM and COM+), Java Remote Method Invocation (RMI), and .NET Remoting are some major examples which fall under Distributed object infrastructure.
- **Active objects** – Programming models based on active objects comprise by definition the presence of instances, whether they are agent or objects, despite the availability of requests. It means, that objects having particular control thread, which enables them to convey their activity. These models sometimes make manual use of messages to encounter the execution of functions and a more complex and a more complex semantics is attached to the messages.
- **Web Services** – Web service technology delivers an approach of the RPC concept over the HTTP, thus enabling the communication of components that are evolved with numerous technologies. A web service is revealed as a remote object maintained on a Web server, and method invocations are transformed in

HTTP requests wrapped with the help of specific protocol. It is necessary to observe that the concept of message is a basic abstraction of inter-process communication and it is utilized either implicitly or explicitly.

communication between child and parent processes is done using kill() and signal(), fork() system call.

- **fork()** creates the child process from the parent. The pid can be checked to decide whether it is the child (if pid == 0) or the parent (pid = child process id).
- The parent can then send messages to child using the pid and kill().
- The child picks up these signals with signal() and calls appropriate functions.

Example of how 2 processes can talk to each other using kill() and signal():

6.15 C

```
// C program to implement sighup(), sigint()
// and sigquit() signal functions
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
    int pid;

    /* get child process */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { /* child */
        signal(SIGUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for (;;) {
            ; /* loop for ever */
    }
}
```

```
else /* parent */
{ /* pid hold id of child */
    printf("\nPARENT: sending SIGHUP\n\n");
    kill(pid, SIGHUP);

    sleep(3); /* pause for 3 secs */
    printf("\nPARENT: sending SIGINT\n\n");
    kill(pid, SIGINT);

    sleep(3); /* pause for 3 secs */
    printf("\nPARENT: sending SIGQUIT\n\n");
    kill(pid, SIGQUIT);
    sleep(3);
}

// sighup() function definition
void sighup()

{
    signal(SIGHUP, sighup); /* reset signal */
    printf("CHILD: I have received a SIGHUP\n");
}

// sigint() function definition
void sigint()

{
    signal(SIGINT, sigint); /* reset signal */
    printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

Output:

```
sahil0612@sahil0612-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
sahil0612@sahil0612-System-Product-Name:~/Desktop$ ./a.out

PARENT: sending SIGHUP

CHILD: I have received a SIGHUP

PARENT: sending SIGINT

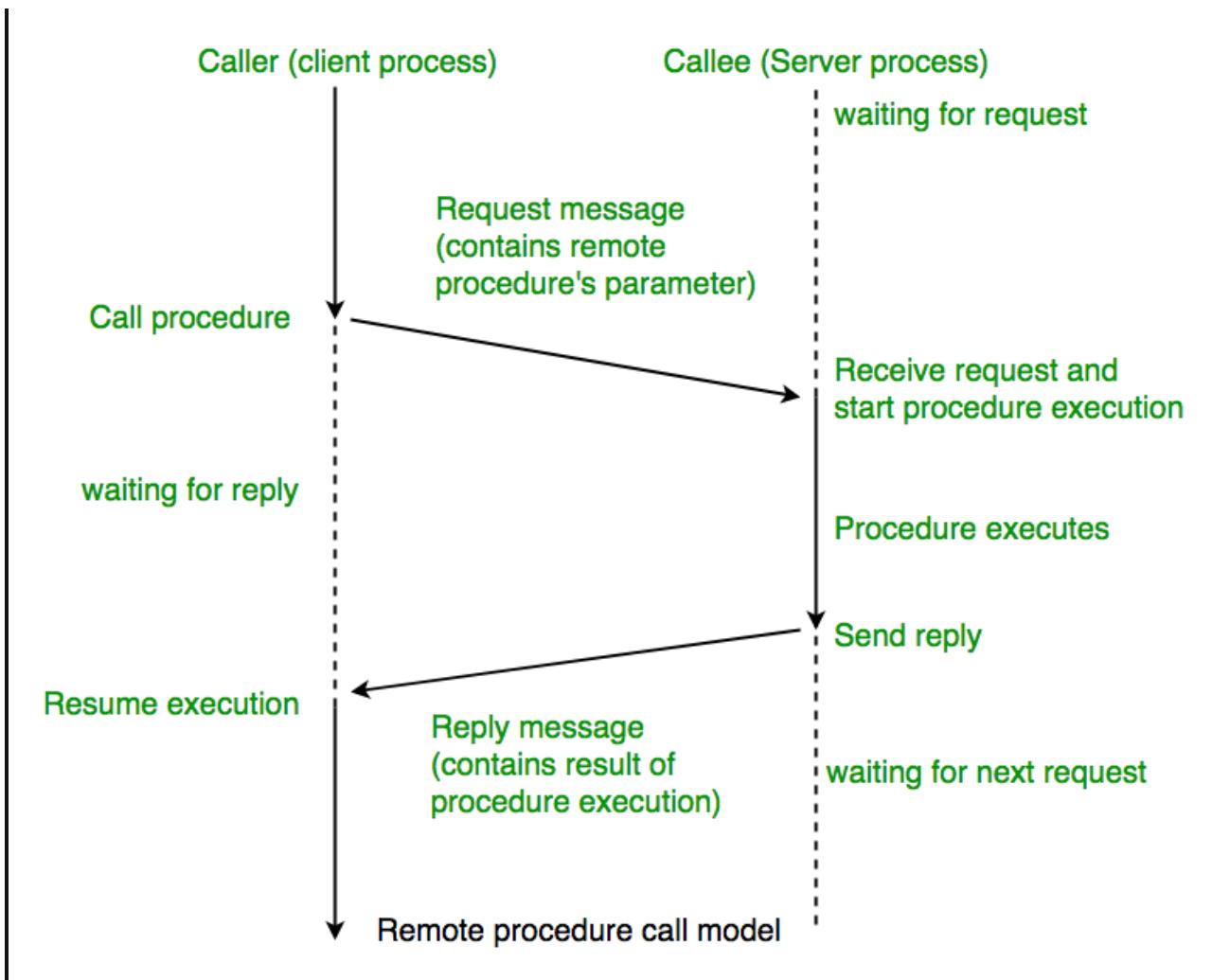
CHILD: I have received a SIGINT

PARENT: sending SIGQUIT

My Parent has Killed me!!!
□
```

Remote Procedure Call (RPC) is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.

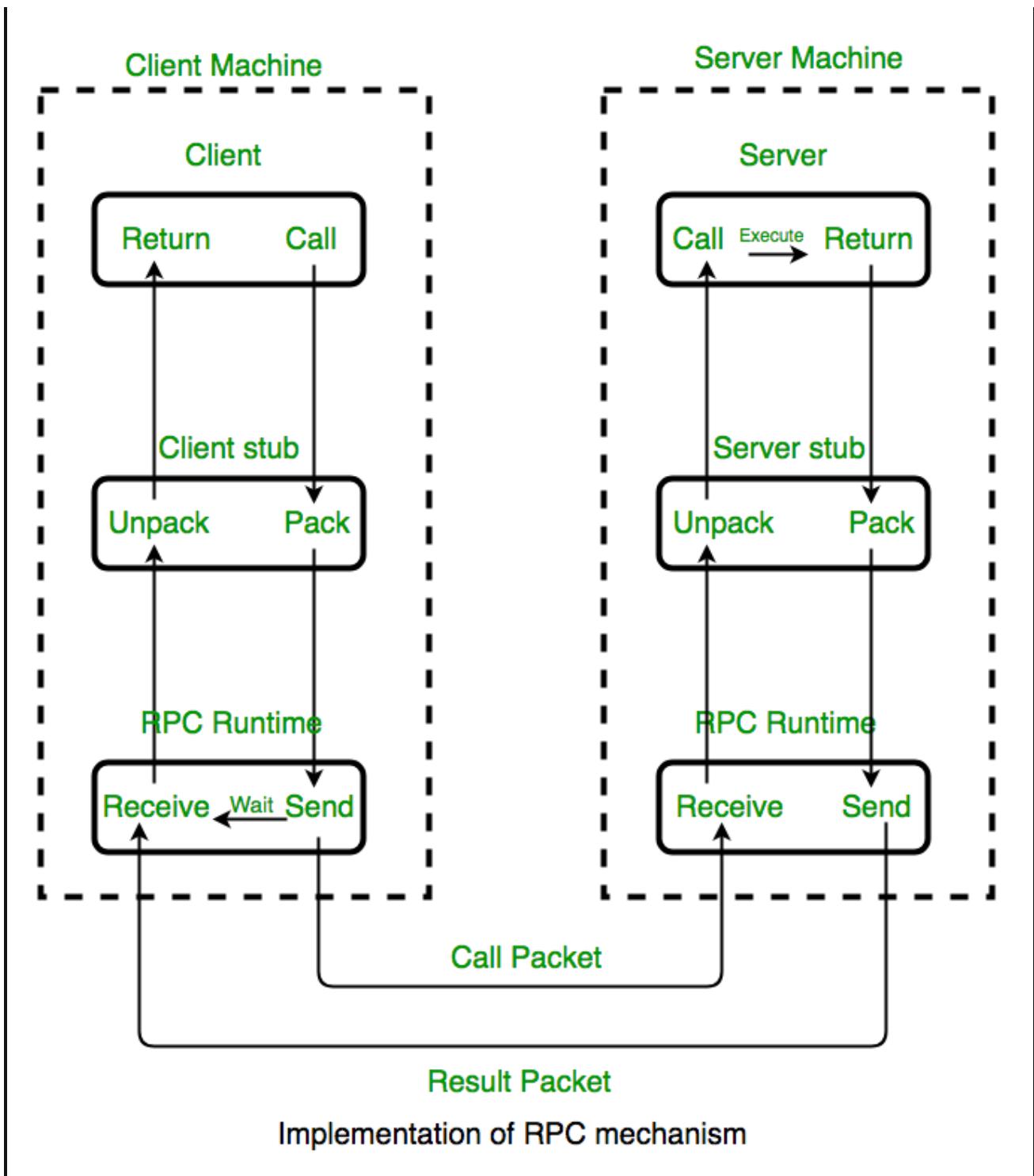
When making a Remote Procedure Call:



1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

NOTE: RPC is especially well suited for client-server (**e.g. query-response**) interaction in which the flow of control **alternates between the caller and callee**. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



The following steps take place during a RPC :

1. A client invokes a **client stub procedure**, passing parameters in the usual way.
The client stub resides within the client's own address space.
2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (**e.g., via a normal procedure call return**), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

Key Considerations for Designing and Implementing RPC Systems are:

- **Security:** Since RPC involves communication over the network, security is a major concern. Measures such as authentication, encryption, and authorization must be implemented to prevent unauthorized access and protect sensitive data.
- **Scalability:** As the number of clients and servers increases, the performance of the RPC system must not degrade. Load balancing techniques and efficient resource utilization are important for scalability.
- **Fault tolerance:** The RPC system should be resilient to network failures, server crashes, and other unexpected events. Measures such as redundancy, failover, and graceful degradation can help ensure fault tolerance.
- **Standardization:** There are several RPC frameworks and protocols available, and it is important to choose a standardized and widely accepted one to ensure interoperability and compatibility across different platforms and programming languages.
- **Performance tuning:** Fine-tuning the RPC system for optimal performance is important. This may involve optimizing the network protocol, minimizing the data transferred over the network, and reducing the latency and overhead associated with RPC calls.

RPC ISSUES :

Issues that must be addressed:

1. RPC Runtime:

RPC run-time system is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**

2. Stub:

The function of the stub is to **provide transparency to the programmer-written application code.**

- **On the client side**, the stub handles the interface between the client's local procedure call and the run-time system, marshalling and unmarshalling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.
- **On the server side**, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

3. Binding: How does the client know who to call, and where the service resides?

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Binding consists of two parts:

- Naming:
 - Locating:
1. **A Server** having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.
 2. **A Client** must import an (exported) interface before communication can begin.

4. The call semantics associated with RPC :

It is mainly classified into following choices-

- **Retry request message –**
Whether to retry sending a request message when a server has failed or the receiver didn't receive the message.
- **Duplicate filtering –**
Remove the duplicate server requests.
- **Retransmission of results –**
To resend lost messages without re-executing the operations at the server side.

ADVANTAGES :

1. RPC provides **ABSTRACTION** i.e message-passing nature of network communication is hidden from the user.
2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.
3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.
4. With RPC code re-writing / re-developing effort is minimized.
5. Process-oriented and thread oriented models supported by RPC.

A Remote Procedure Call (RPC) is a powerful concept in distributed computing where a program can cause a procedure (subroutine) to execute in another address space (commonly on another machine) as if it were a local procedure call without the programmer explicitly coding the details for remote communication. It abstracts away the complexities of communication between different processes or machines.

Interface Definition Language (IDL) is often used to define the interface between the client and the server. IDL describes the data structures and procedures that can be used remotely. Here's a simple example in C using the Open Network Computing (ONC) RPC mechanism:

Let's say we want to create a remote calculator service that provides basic arithmetic operations.

1. Define the Interface in IDL (calculator.x):

```
idlCopy code
// calculator.x const int MAX_STRING = 256; struct operands { int operand1; int operand2; }; program
CALCULATOR_PROG { version CALCULATOR_VERS { int ADD(operands) = 1; int SUBTRACT(operands) = 2; int
MULTIPLY(operands) = 3; int DIVIDE(operands) = 4; } = 1; } = 0x31230000;
```

This IDL file defines a simple calculator program with four operations: ADD, SUBTRACT, MULTIPLY, and DIVIDE.

2. Run the IDL Compiler:

Use the rpcgen tool to generate the necessary client and server files from the IDL file.

```
bashCopy code
rpcgen calculator.x
```

This will generate:

- **calculator.h**: Header file with data structure and function prototypes.
- **calculator_svc.c**: Server-side code.
- **calculator_clnt.c**: Client-side code.
- **calculator_xdr.c**: XDR (External Data Representation) routines.

3. Implement the Server (server.c):

```
cCopy code
// server.c #include "calculator.h" int *add_1_svc(operands *args, struct svc_req *rqstp) { static int result;
result = args->operand1 + args->operand2; return &result; } // Implement SUBTRACT, MULTIPLY, and DIVIDE
similarly
```

4. Implement the Client (client.c):

```
cCopy code
// client.c #include <stdio.h> #include "calculator.h" int main(int argc, char *argv[]) { CLIENT *cl; operands
operands_arg; int *result; if (argc != 4) { fprintf(stderr, "Usage: %s <hostname> <operand1> <operand2>\n",
argv[0]); exit(1); } operands_arg.operand1 = atoi(argv[2]); operands_arg.operand2 = atoi(argv[3]); cl =
clnt_create(argv[1], CALCULATOR_PROG, CALCULATOR_VERS, "tcp"); if (cl == NULL) {
clnt_pcreateerror(argv[1]); exit(1); } switch (atoi(argv[2])) { case 1: result = add_1(&operands_arg, cl);
break; // Implement cases for SUBTRACT, MULTIPLY, and DIVIDE // ... default: fprintf(stderr, "Invalid
operation\n"); exit(1); } if (result == NULL) { clnt_perror(cl, argv[1]); exit(1); } printf("Result: %d\n",
*result); clnt_destroy(cl); return 0; }
```

5. Compile and Run:

Compile both the server and client programs.

```
bashCopy code
```

```
gcc server.c calculator_svc.c calculator_xdr.c -o server gcc client.c calculator_clnt.c calculator_xdr.c -o client
```

Run the server and client on different machines or on the same machine with different terminal windows.

```
bashCopy code
```

```
./server
```

In another terminal window:

```
bashCopy code
```

```
./client localhost 1 2
```

This example demonstrates a basic RPC implementation using C and an IDL file. The server provides a set of arithmetic operations, and the client can remotely invoke these operations as if they were local function calls.

explain more clearly as to how idl is written

ChatGPT

Certainly! Interface Definition Language (IDL) is a language-independent way to describe the interface of a software component. It's often used in the context of Remote Procedure Call (RPC) systems to define the communication interface between different parts of a distributed system. Let me break down the IDL file used in the previous example more clearly.

1. Choose a File Extension:

- IDL files typically have the extension `.idl`. In the example, the file is named `calculator.x`.

2. Define Constants:

- You can define constants that will be used in your IDL file. In the example, `MAX_STRING` is defined.

```
idlCopy code
```

3. Define Data Structures:

- Use `struct` to define the data structures that will be used as parameters or return values in your remote procedures. In the example, a structure named `operands` is defined with two integer fields.

```
idlCopy code
```

4. Define the RPC Program:

- Use the `program` keyword to define the RPC program. The program is the logical unit of organization for a set of related procedures. In the example, the program is named `CALCULATOR_PROG`.

idlCopy code

5. Define Program Versions:

- Within the program, you can define different versions using the `version` keyword. Each version can have its own set of procedures. In the example, a version named `CALCULATOR_VERS` is defined.

idlCopy code

6. Define Remote Procedures:

- Use the keyword `int` followed by the procedure name and its parameters to define remote procedures. In the example, four procedures are defined: `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`.

idlCopy code

- The numbers `1`, `2`, `3`, and `4` are procedure numbers. These numbers are used to uniquely identify each procedure within the program.

7. End Definitions:

- Close the definitions using appropriate closing braces.

idlCopy code

- The final number (`0x31230000`) is a version number. It's a unique identifier for the version of the program.

So, in summary, an IDL file defines data structures and a set of remote procedures organized within programs and versions. This definition serves as a contract between the client and server, enabling them to communicate seamlessly in a distributed system. The IDL file is then processed by a tool like `rpcgen` to generate code for the client and server to use in order to make remote procedure calls.



what will be the content of calculator_svc.c

ChatGPT

In the server-side code, `calculator_svc.c`, you need to implement the server-side functions corresponding to the remote procedures defined in your Interface Definition Language (IDL) file. Each remote procedure should have a corresponding implementation in this file. Here's how the `calculator_svc.c` file might look for the example with four arithmetic operations:

cCopy code

```
include "calculator.h" int add_1_svc struct static int
return int subtract_1_svc struct
static int return int
multiply_1_svc struct static int
return int divide_1_svc struct static int if
0 else fprintf stderr "Error: Division by
zero\n" return NULL return
```

Explanation:

- Each function in `calculator_svc.c` corresponds to one of the remote procedures defined in the IDL file (`ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`).
- The function names follow a specific convention: `<procedure_name>_1_svc`. The `_1_svc` suffix is added automatically by the `rpcgen` tool.
- These functions take the input parameters specified in the IDL file (e.g., `operands *args`) and return a pointer to the result.

Remember that this is just the server-side implementation of the remote procedures. The client-side code will invoke these remote procedures using functions generated by `rpcgen`. After implementing the server code, you would compile it along with the client code and run the server in one terminal and the client in another to test the remote procedure calls.



what will be the content of `calculator_clnt.c`

ChatGPT

In the client-side code, `calculator_clnt.c`, you need to implement the client-side functions that will be used to invoke the remote procedures defined in your Interface Definition Language (IDL) file. These client-side functions are automatically generated by the `rpcgen` tool based on the IDL file. Here's how the `calculator_clnt.c` file might look for the example with four arithmetic operations:

Copy code

```
include "calculator.h"
static struct timeval TIMEOUT { 25, 0 } int add_1 static int
memset(char, 0, sizeof(if xdrproc_t caddr_t
xdrproc_t caddr_t) return NULL return
int subtract_1 static int memset(char
0, sizeof(if xdrproc_t caddr_t xdrproc_t
caddr_t) return NULL return int
multiply_1 static int memset(char 0, sizeof
if xdrproc_t caddr_t xdrproc_t caddr_t
return NULL return int divide_1
static int memset(char 0, sizeof if
xdrproc_t caddr_t xdrproc_t caddr_t
return NULL return)
```

Explanation:

- Each function in `calculator_clnt.c` corresponds to one of the remote procedures defined in the IDL file (`ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`).
- The function names match those in the IDL file.
- These functions are used by the client to make remote procedure calls. They internally use the `cint_call` function, which is part of the RPC library, to send requests to the server and receive the results.
- The `xdr_operands` and `xdr_int` are functions that handle the serialization and deserialization of data structures. They are also generated by `rpcgen` based on the IDL file.

After implementing the client code, you would compile it along with the server code and run the server in one terminal and the client in another to test the remote procedure calls.



what will be the content of calculator_xdr.c

ChatGPT

In the Remote Procedure Call (RPC) mechanism, External Data Representation (XDR) is used to standardize the encoding and decoding of data structures for communication between different machines or processes. The `calculator_xdr.c` file contains the XDR routines generated by the `rpcgen` tool based on the Interface Definition Language (IDL) file. These routines handle the serialization and deserialization of data structures.

Here's how the `calculator_xdr.c` file might look for the example with four arithmetic operations:

Copy code

```
include "calculator.h" bool_t xdr_operands
return if
      return return
bool_t xdr_int    int   if
      return return
```

Explanation:

- The `xdr_operands` function is responsible for encoding and decoding the `operands` structure.
- The `xdr_int` function is responsible for encoding and decoding integer values.
- These functions use the `XDR` data stream to serialize and deserialize the data structures. The `XDR` data stream is a standardized format that allows data to be represented in a machine-independent way for communication between systems.

These XDR routines are essential for correctly encoding data before it is sent over the network and decoding it on the receiving side. The `rpcgen` tool generates these routines based on the data types defined in the IDL file.

When compiling your project, you typically include these XDR routines along with the server and client files to ensure that the data can be properly serialized and deserialized during remote procedure calls.

`clnt_call` is a function provided by the RPC (Remote Procedure Call) library in C, and it is used to invoke remote procedures on a server. This function is automatically generated by the `rpcgen` tool based on the Interface Definition Language (IDL) file and is part of the client-side code.

Here is the basic syntax of the `clnt_call` function:

Copy code

```
int clnt_call(CLIENT *clnt, u_long procnum, xdrproc_t inproc, caddr_t in, xdrproc_t outproc, caddr_t out, struct timeval timeout);
```

- `clnt`: A pointer to the client handle, which is obtained through functions like `clnt_create`.

- **procnum**: The procedure number corresponding to the remote procedure to be called. This number is specific to each remote procedure and is defined in the IDL file.
- **inproc**: A pointer to the XDR (External Data Representation) encoding function for the input parameters of the remote procedure.
- **in**: A pointer to the input parameters of the remote procedure.
- **outproc**: A pointer to the XDR decoding function for the output result of the remote procedure.
- **out**: A pointer to the location where the output result of the remote procedure will be stored.
- **timeout**: A structure specifying the maximum amount of time the client should wait for the remote procedure call to complete.

Here is a breakdown of the parameters:

- **cInt**: Represents the client's connection to the server. It contains information about the server's location, transport protocol, and other details.
- **procnum**: Identifies the specific remote procedure to be called. It corresponds to the procedure number defined in the IDL file.
- **inproc** and **in**: These parameters specify the encoding function and input parameters for the remote procedure call. The **xdrproc_t** type represents an XDR encoding or decoding function, and **caddr_t** is a character address, essentially a pointer to data.
- **outproc** and **out**: These parameters specify the decoding function and the location to store the result of the remote procedure call.
- **timeout**: Specifies the maximum time the client is willing to wait for the remote procedure call to complete. It's a struct of type **struct timeval**, where you can set the seconds and microseconds.

The **cInt_call** function initiates the remote procedure call, handles the communication with the server, and returns the result to the client.

It's important to note that the **cInt_call** function is part of the lower-level RPC API, and in more modern distributed systems, developers often use higher-level abstractions and libraries for remote communication, such as gRPC or RESTful APIs.

In the Operating System, **Mutexes and Semaphores** are kernel resources that provide synchronization services (also known as synchronization primitives). Synchronization is required when multiple processes are executing concurrently, to avoid conflicts between processes using shared resources.

6.16 Mutex

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for [Mutual Exclusion](#) Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

Mutex uses a priority inheritance mechanism to avoid [priority inversion](#) issues. The priority inheritance mechanism keeps higher-priority processes in the blocked state for the minimum possible time. However, this cannot avoid the priority inversion problem, but it can reduce its effect up to an extent.

Advantages of Mutex

- No race condition arises, as only one process is in the critical section at a time.
- Data remains consistent and it helps in maintaining integrity.
- It's a simple locking mechanism that can be obtained by a process before entering into a critical section and released while leaving the critical section.

Disadvantages of Mutex

- If after entering into the critical section, the thread sleeps or gets preempted by a high-priority process, no other thread can enter into the critical section. This can lead to starvation.
- When the previous thread leaves the critical section, then only other processes can enter into it, there is no other mechanism to lock or unlock the critical section.
- Implementation of mutex can lead to busy waiting that leads to the wastage of the CPU cycle.

Using Mutex

The producer-consumer problem: Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

Solution- A mutex provides mutual exclusion, either [producer or consumer](#) can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa. At any point in time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.

Note- The content is a generalized explanation. Practical details vary with implementation.

6.17 Semaphore

A semaphore is a non-negative integer variable that is shared between various threads. Semaphore works upon signaling mechanism, in this a thread can be signaled by another thread. Semaphore uses two atomic operations for process synchronisation:

- Wait (P)
- Signal (V)

Advantages of Semaphore

- Multiple threads can access the critical section at the same time.
- [Semaphores](#) are machine-independent.
- Only one process will access the critical section at a time, however, multiple threads are allowed.
- Semaphores are machine-independent, so they should be run over [microkernel](#).
- Flexible resource management.

Disadvantages of Semaphore

- It has [priority inversion](#).
- Semaphore's operation (Wait, Signal) must be implemented in the correct manner to avoid deadlock.
- It leads to a loss of modularity, so semaphores can't be used for large-scale systems.
- Semaphore is prone to programming error and this can lead to deadlock or violation of mutual exclusion property.
- Operating System has to track all the calls to wait and signal operations.

6.17.1.1 Using Semaphore

The [producer-consumer](#) problem: Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

Solution- A semaphore is a generalized **mutex**. In lieu of a single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A [semaphore](#) can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Note: The content is a generalized explanation. Practical details vary with implementation.

6.18 Difference Between Mutex and Semaphore

Mutex	Semaphore
A mutex is an object.	A semaphore is an integer.
Mutex works upon the locking mechanism.	Semaphore uses signaling mechanism
Operations on mutex: <ul style="list-style-type: none">• Lock• Unlock	Operation on semaphore: <ul style="list-style-type: none">• Wait• Signal
Mutex doesn't have any subtypes.	Semaphore is of two types: <ul style="list-style-type: none">• Counting Semaphore• Binary Semaphore
A mutex can only be modified by the process that is requesting or releasing a resource.	Semaphore work with two atomic operations (Wait, signal) which can modify it.

Mutex	Semaphore
If the mutex is locked then the process needs to wait in the process queue, and mutex can only be accessed once the lock is released.	If the process needs a resource, and no resource is free. So, the process needs to perform a wait operation until the semaphore value is greater than zero.

6.19 Misconception

There is an ambiguity between binary semaphore and mutex. We might have come across that a mutex is a binary semaphore. But it is not! The purposes of mutex and semaphore are different. Maybe, due to similarity in their implementation a mutex would be referred to as a binary semaphore.

A mutex is a locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

Semaphore is a signaling mechanism ("I am done, you can carry on" kind of signal). For example, if you are listening to songs (assume it as one task) on your mobile phone and at the same time, your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives. When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0 . If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

Semaphores are of two types:

1. **Binary Semaphore –**

This is also known as a mutex lock. It can have only two values – 0 and 1. Its

value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

2. Counting Semaphore –

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Now let us see how it does so.

First, look at two operations that can be used to access and change the value of the semaphore variable.

```
P(Semaphore s){  
    while(S == 0); /* wait until s=0 */  
    s=s-1;  
}  
  
V(Semaphore s){  
    s=s+1;  
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

Some points regarding P and V operation:

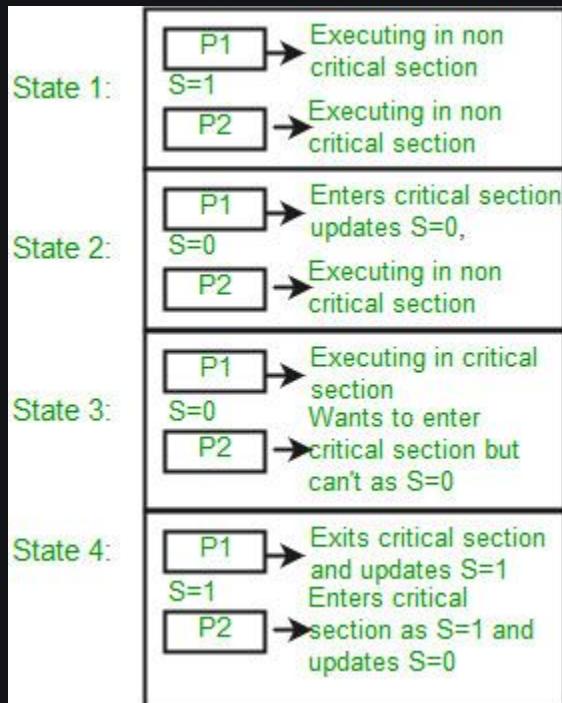
1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization. See the below image. The critical section of Process P is in between P and V operation.

Process P

```
// Some code  
P(s);  
// critical section  
V(s);  
// remainder section
```

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until $s > 0$, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.

This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.



```
#include <stdio.h>  
#include<stdlib.h>  
#include<sys/queue.h>
```

```

struct semaphore{
    Queue<process> q;
    int value;
};

void P(struct semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        s.q.push(P);
        sleep();
    }
}

void V(Semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
    }
    else {

        // Get a process from the Waiting Queue
        Process p = q.front();
        // Remove the process from waiting
        q.pop();
        wakeup(p);
    }
}
int main() {
    printf("This is Hemish!!");

    // This code is Contributed by Himesh Singh Chauhan
    return 0;
}

```

Certainly! Let's consider another example of using counting semaphores in C to solve the classic producer-consumer problem. In this scenario, multiple producer threads add items to a shared buffer, and multiple consumer threads retrieve and process these items. The counting semaphore is used to regulate access to the shared buffer.

 Copy code

```

#include <stdio.h> #include <pthread.h> #include <semaphore.h> #define BUFFER_SIZE 5 #define
NUM_PRODUCERS 2 #define NUM_CONSUMERS 3 // Shared circular buffer and counting semaphores int
buffer[BUFFER_SIZE]; sem_t empty_slots, filled_slots; pthread_mutex_t buffer_mutex =
PTHREAD_MUTEX_INITIALIZER; void *producer(void *arg) { int item = 1; // Item to be produced while ([1]) {
// Wait until there is an empty slot in the buffer sem_wait(&empty_slots); // Critical section: Add item to the
buffer pthread_mutex_lock(&buffer_mutex); printf("Producing item %d\n", item); buffer[item % BUFFER_SIZE]
= item++; pthread_mutex_unlock(&buffer_mutex); // Signal that a slot is filled in the buffer
sem_post(&filled_slots); // Simulate some work usleep(200000); // Sleep for 200 milliseconds } return NULL;
} void *consumer(void *arg) { while ([1]) { // Wait until there is a filled slot in the buffer
sem_wait(&filled_slots); // Critical section: Remove item from the buffer pthread_mutex_lock(&buffer_mutex);
int item = buffer[item % BUFFER_SIZE]; printf("Consuming item %d\n", item);
pthread_mutex_unlock(&buffer_mutex); // Signal that an empty slot is available in the buffer
sem_post(&empty_slots); // Simulate some work usleep(300000); // Sleep for 300 milliseconds } return
NULL; } int main() { // Initialize counting semaphores sem_init(&empty_slots, 0, BUFFER_SIZE); // Number of
empty slots in the buffer sem_init(&filled_slots, 0, 0); // Number of filled slots in the buffer // Create producer
and consumer threads pthread_t producer_threads[NUM_PRODUCERS]; pthread_t
consumer_threads[NUM_CONSUMERS]; for (int i = 0; i < NUM_PRODUCERS; ++i) {
pthread_create(&producer_threads[i], NULL, producer, NULL); } for (int i = 0; i < NUM_CONSUMERS; ++i) {
pthread_create(&consumer_threads[i], NULL, consumer, NULL); } // Wait for threads to finish (this won't
happen in this example because the threads run indefinitely) for (int i = 0; i < NUM_PRODUCERS; ++i) {
pthread_join(producer_threads[i], NULL); } for (int i = 0; i < NUM_CONSUMERS; ++i) {
pthread_join(consumer_threads[i], NULL); } // Clean up semaphores sem_destroy(&empty_slots);
sem_destroy(&filled_slots); return 0; }

```

In this example:

1. There's a shared circular buffer (`buffer`) of a fixed size.
2. Two counting semaphores, `empty_slots` and `filled_slots`, are used to control access to the buffer.
3. The `producer` function adds items to the buffer when there is an empty slot, and the `consumer` function removes items when there is a filled slot.
4. The producer and consumer threads run concurrently, and the counting semaphores ensure that producers wait when the buffer is full and consumers wait when the buffer is empty.

This example demonstrates how counting semaphores can be used to coordinate multiple threads accessing a shared resource in a producer-consumer scenario. The mutual exclusion provided by the semaphores helps prevent race conditions and ensures the correct synchronization of producers and consumers.

In one line, *Priority Inversion* is a **problem** while *Priority Inheritance* is a **solution**. *Priority Inversion* means that the priority of tasks gets inverted and *Priority Inheritance* means that the priority of tasks gets inherited. Both of these phenomena happen in priority scheduling. Basically, in *Priority Inversion*, the higher priority task (H) ends up waiting for the middle priority task (M) when H is sharing a critical section with the lower priority task (L) and L is

already in the critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion. One of the solutions to this problem is *Priority Inheritance*. In *Priority Inheritance*, when L is in the critical section, L inherits the priority of H at the time when H starts pending for the critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of the critical section.

Let us see the differences in a tabular form:

	Priority Inversion	Priority Inheritance
1.	In priority inversion , a higher-priority process is preempted by a lower-priority process.	It is a method that is used to eliminate the problems of Priority inversion.
2.	It is the inversion of the priorities of the two processes	With the help of this, a process scheduling algorithm increases the priority of a process, to the maximum priority of any other process waiting for any resource.
3.	It can cause a system to malfunction in our system.	Priority inheritance can lead to poorer worst-case behavior when there are nested locks.
4.	Priority inversions can lead to the implementation of corrective measures.	Priority inheritance can be implemented such that there is no penalty when the locks do not contend,
5.	To deal with the problem of priority inversion we can have several techniques such as Priority ceiling, Random boosting, etc.	It is the basic technique at the application level for managing priority inversion.

Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.

1. A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures. The procedures provided by a monitor ensure that only one process can access the shared

- resource at any given time, and that processes waiting for the resource are suspended until it becomes available.
2. Monitors are used to simplify the implementation of concurrent programs by providing a higher-level abstraction that hides the details of synchronization. Monitors provide a structured way of sharing data and synchronization information, and eliminate the need for complex synchronization primitives such as semaphores and locks.
 3. The key advantage of using monitors for process synchronization is that they provide a simple, high-level abstraction that can be used to implement complex concurrent systems. Monitors also ensure that synchronization is encapsulated within the module, making it easier to reason about the correctness of the system.

However, monitors have some limitations. For example, they can be less efficient than lower-level synchronization primitives such as semaphores and locks, as they may involve additional overhead due to their higher-level abstraction. Additionally, monitors may not be suitable for all types of synchronization problems, and in some cases, lower-level primitives may be required for optimal performance.

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}

}
```

Syntax of Monitor

Syntax:

Problem: Given 2 processes i and j, you need to write a program that can guarantee mutual exclusion between the two without any additional hardware support.

Solution: There can be multiple ways to solve this problem, but most of them require additional hardware support. The simplest and the most popular way to do this is by using Peterson's Algorithm for mutual Exclusion. It was developed by Peterson in 1981 though the initial work in this direction was done by Theodorus Jozef Dekker who came up with **Dekker's algorithm** in 1960, which was later refined by Peterson and came to be known as **Peterson's Algorithm**.

Basically, Peterson's algorithm provides guaranteed mutual exclusion by using only the shared memory. It uses two ideas in the algorithm:

1. Willingness to acquire lock.
2. Turn to acquire lock.

6.19.1.1 *Explanation:*

The idea is that first a thread expresses its desire to acquire a lock and sets **flag[self] = 1** and then gives the other thread a chance to acquire the lock. If the thread desires to acquire the lock, then, it gets the lock and passes the chance to the 1st thread. If it does not desire to get the lock then the while loop breaks and the 1st thread gets the chance.

Implementation in C language

6.20 C

```
// Filename: peterson_spinlock.c
// Use below command to compile:
// gcc -pthread peterson_spinlock.c -o peterson_spinlock

#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

int flag[2];
int turn;
const int MAX = 1e9;
int ans = 0;

void lock_init()
{
    // Initialize lock by resetting the desire of
    // both the threads to acquire the locks.
    // And, giving turn to one of them.
    flag[0] = flag[1] = 0;
    turn = 0;
}

// Executed before entering critical section
void lock(int self)
{
```

```
// Set flag[self] = 1 saying you want to acquire lock
flag[self] = 1;

// But, first give the other thread the chance to
// acquire lock
turn = 1-self;

// Wait until the other thread loses the desire
// to acquire lock or it is your turn to get the lock.
while (flag[1-self]==1 && turn==1-self) ;
}

// Executed after leaving critical section
void unlock(int self)
{
    // You do not desire to acquire lock in future.
    // This will allow the other thread to acquire
    // the lock.
    flag[self] = 0;
}

// A Sample function run by two threads created
// in main()
void* func(void *s)
{
    int i = 0;
    int self = (int *)s;
    printf("Thread Entered: %d\n", self);

    lock(self);

    // Critical section (Only one thread
    // can enter here at a time)
    for (i=0; i<MAX; i++)
        ans++;

    unlock(self);
}

// Driver code
int main()
{
    // Initialized the lock then fork 2 threads
    pthread_t p1, p2;
```

```

lock_init();

// Create two threads (both run func)
pthread_create(&p1, NULL, func, (void*)0);
pthread_create(&p2, NULL, func, (void*)1);

// Wait for the threads to end.
pthread_join(p1, NULL);
pthread_join(p2, NULL);

printf("Actual Count: %d | Expected Count: %d\n",
       ans, MAX*2);

return 0;
}

```

Problem: Given 2 process i and j, you need to write a program that can guarantee mutual exclusion between the two without any additional hardware support.

We strongly recommend to refer below basic solution discussed in previous article.

[Peterson's Algorithm for Mutual Exclusion | Set 1](#)

We would be resolving 2 issues in the previous algorithm.

6.20.1.1 *Wastage of CPU clock cycles*

In layman terms, when a thread was waiting for its turn, it ended in a long while loop which tested the condition millions of times per second thus doing unnecessary computation. There is a better way to wait, and it is known as “yield”.

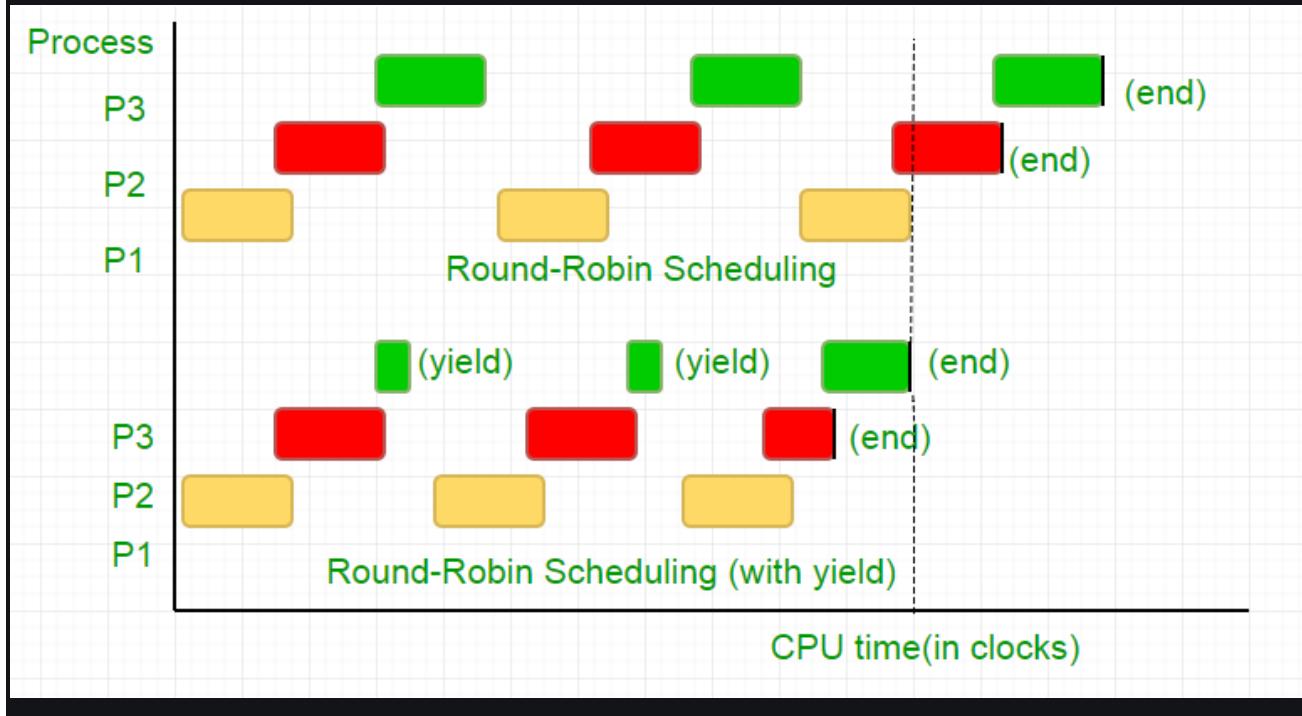
To understand what it does, we need to dig deep into how the Process scheduler works in Linux. The idea mentioned here is a simplified version of the scheduler, the actual implementation has lots of complications.

Consider the following example,

There are three processes, P1, P2 and P3. Process P3 is such that it has a while loop similar to the one in our code, doing not so useful computation, and it exists from the loop only when P2 finishes its execution. The scheduler puts all of them in a round robin queue. Now, say the clock speed of processor is 1000000/sec, and it allocates 100 clocks to each process in each iteration. Then, first P1 will be run for 100 clocks (0.0001 seconds), then P2(0.0001 seconds) followed by P3(0.0001 seconds), now since there are no more processes, this cycle repeats until P2 ends and then followed by P3’s execution and eventually its termination.

This is a complete waste of the 100 CPU clock cycles. To avoid this, we mutually give up the CPU time slice, i.e. yield, which essentially ends this time slice and the scheduler picks up the next process to run. Now, we test our condition once, then we give up the CPU. Considering our test takes 25 clock cycles, we save 75% of our computation in a time slice.

To put this graphically,



Considering the processor clock speed as 1MHz this is a lot of saving!. Different distributions provide different function to achieve this functionality. Linux provides `sched_yield()`.

- C

```
void lock(int self)
{
    flag[self] = 1;
    turn = 1-self;

    while (flag[1-self] == 1 &&
           turn == 1-self)

        // Only change is the addition of
        // sched_yield() call
        sched_yield();
}
```

6.20.1.2 Memory fence.

The code in earlier tutorial might have worked on most systems, but it was not 100% correct. The logic was perfect, but most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but can cause

unpredictable behaviour in concurrent programs.

Consider this example,

- C

```
while (f == 0);

// Memory fence required here
print x;
```

In the above example, the compiler considers the 2 statements as independent of each other and thus tries to increase the code efficiency by re-ordering them, which can lead to problems for concurrent programs. To avoid this we place a memory fence to give hint to the compiler about the possible relationship between the statements across the barrier.

So the order of statements,

has to be exactly the same in order for the lock to work, otherwise it will end up in a deadlock condition.

To ensure this, compilers provide a instruction that prevent ordering of statements across this barrier. In case of gcc, its `__sync_synchronize()`.

So the modified code becomes,

Full Implementation in C:

- C

```
// Filename: peterson_yieldlock_memoryfence.c
// Use below command to compile:
// gcc -pthread peterson_yieldlock_memoryfence.c -o peterson_yieldlock_memoryfence

#include<stdio.h>
#include<pthread.h>
#include "mythreads.h"

int flag[2];
int turn;
const int MAX = 1e9;
int ans = 0;
```

```
void lock_init()
{
    // Initialize lock by resetting the desire of
    // both the threads to acquire the locks.
    // And, giving turn to one of them.
    flag[0] = flag[1] = 0;

    turn = 0;
}

// Executed before entering critical section
void lock(int self)
{
    // Set flag[self] = 1 saying you want
    // to acquire lock
    flag[self]=1;

    // But, first give the other thread the
    // chance to acquire lock
    turn = 1-self;

    // Memory fence to prevent the reordering
    // of instructions beyond this barrier.
    __sync_synchronize();

    // Wait until the other thread loses the
    // desire to acquire lock or it is your
    // turn to get the lock.
    while (flag[1-self]==1 && turn==1-self)

        // Yield to avoid wastage of resources.
        sched_yield();
}

// Executed after leaving critical section
void unlock(int self)
{
    // You do not desire to acquire lock in future.
    // This will allow the other thread to acquire
    // the lock.
    flag[self]=0;
}

// A Sample function run by two threads created
```

```

// in main()
void* func(void *s)
{
    int i = 0;
    int self = (int *)s;
    printf("Thread Entered: %d\n",self);
    lock(self);

    // Critical section (Only one thread
    // can enter here at a time)
    for (i=0; i<MAX; i++)
        ans++;

    unlock(self);
}

// Driver code
int main()
{
    pthread_t p1, p2;

    // Initialize the lock
    lock_init();

    // Create two threads (both run func)
    Pthread_create(&p1, NULL, func, (void*)0);
    Pthread_create(&p2, NULL, func, (void*)1);

    // Wait for the threads to end.
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);

    printf("Actual Count: %d | Expected Count:"
           " %d\n",ans,MAX*2);

    return 0;
}

```

- C

// C program to implement Peterson's Algorithm
// for producer-consumer problem.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>

```

```

#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#define _BSD_SOURCE
#include <sys/time.h>

#define BSIZE 8 // Buffer size
#define PWT 2 // Producer wait time limit
#define CWT 10 // Consumer wait time limit
#define RT 10 // Program run-time in seconds

int shmid1, shmid2, shmid3, shmid4;
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;
bool* SHM1;
int* SHM2;
int* SHM3;

int myrand(int n) // Returns a random number between 1 and n
{
    time_t t;
    srand((unsigned)time(&t));
    return (rand() % n + 1);
}

int main()
{
    shmid1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660); // flag
    shmid2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660); // turn
    shmid3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
    shmid4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660); // time stamp

    if (shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {
        perror("Main shmget error: ");
        exit(1);
    }
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    int ix = 0;
    while (ix < BSIZE) // Initializing buffer
        SHM3[ix++] = 0;

    struct timeval t;
    time_t t1, t2;
    gettimeofday(&t, NULL);
    t1 = t.tv_sec;

    int* state = (int*)shmat(shmid4, NULL, 0);
    *state = 1;
    int wait_time;

    int i = 0; // Consumer
    int j = 1; // Producer

    if (fork() == 0) // Producer code
    {
        SHM1 = (bool*)shmat(shmid1, NULL, 0);
        SHM2 = (int*)shmat(shmid2, NULL, 0);
        SHM3 = (int*)shmat(shmid3, NULL, 0);
        if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {

```

```

        perror("Producer shmat error: ");
        exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;
    int* buf = SHM3;
    int index = 0;

    while (*state == 1) {
        flag[j] = true;
        printf("Producer is ready now.\n\n");
        *turn = i;
        while (flag[i] == true && *turn == i)
            ;

        // Critical Section Begin
        index = 0;
        while (index < BSIZE) {
            if (buf[index] == 0) {
                int tempo = myrand(BSIZE * 3);
                printf("Job %d has been produced\n", tempo);
                buf[index] = tempo;
                break;
            }
            index++;
        }
        if (index == BSIZE)
            printf("Buffer is full, nothing can be produced!!!\n");
        printf("Buffer: ");
        index = 0;
        while (index < BSIZE)
            printf("%d ", buf[index++]);
        printf("\n");
        // Critical Section End

        flag[j] = false;
        if (*state == 0)
            break;
        wait_time = myrand(PWT);
        printf("Producer will wait for %d seconds\n\n", wait_time);
        sleep(wait_time);
    }
    exit(0);
}

if (fork() == 0) // Consumer code
{
    SHM1 = (bool*)shmat(shmid1, NULL, 0);
    SHM2 = (int*)shmat(shmid2, NULL, 0);
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
        perror("Consumer shmat error:");
        exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;

```

```

int* buf = SHM3;
int index = 0;
flag[i] = false;
sleep(5);
while (*state == 1) {
    flag[i] = true;
    printf("Consumer is ready now.\n\n");
    *turn = j;
    while (flag[j] == true && *turn == j)
        ;
}

// Critical Section Begin
if (buf[0] != 0)
    printf("Job %d has been consumed\n", buf[0]);
    buf[0] = 0;
    index = 1;
    while (index < BSIZE) // Shifting remaining jobs forward
    {
        buf[index - 1] = buf[index];
        index++;
    }
    buf[index - 1] = 0;
} else
    printf("Buffer is empty, nothing can be consumed!!!\n");
printf("Buffer: ");
index = 0;
while (index < BSIZE)
    printf("%d ", buf[index++]);
printf("\n");
// Critical Section End

flag[i] = false;
if (*state == 0)
    break;
wait_time = myrand(CWT);
printf("Consumer will sleep for %d seconds\n\n", wait_time);
sleep(wait_time);
}
exit(0);
}
// Parent process will now for RT seconds before causing child to terminate
while (1) {
    gettimeofday(&t, NULL);
    t2 = t.tv_sec;
    if (t2 - t1 > RT) // Program will exit after RT seconds
    {
        *state = 0;
        break;
    }
}
// Waiting for both processes to exit
wait();
wait();
printf("The clock ran out.\n");
return 0;
}

#include <stdio.h>
#include <pthread.h>

```

```

#include <semaphore.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 3

// Shared circular buffer and counting semaphores
int buffer[BUFFER_SIZE];
sem_t empty_slots, filled_slots;
pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {
    int item = 1; // Item to be produced

    while (1) {
        // Wait until there is an empty slot in the buffer
        sem_wait(&empty_slots);

        // Critical section: Add item to the buffer
        pthread_mutex_lock(&buffer_mutex);
        printf("Producing item %d\n", item);
        buffer[item % BUFFER_SIZE] = item++;
        pthread_mutex_unlock(&buffer_mutex);

        // Signal that a slot is filled in the buffer
        sem_post(&filled_slots);

        // Simulate some work
        usleep(200000); // Sleep for 200 milliseconds
    }

    return NULL;
}

void *consumer(void *arg) {
    while (1) {
        // Wait until there is a filled slot in the buffer
        sem_wait(&filled_slots);

        // Critical section: Remove item from the buffer
        pthread_mutex_lock(&buffer_mutex);
        int item = buffer[item % BUFFER_SIZE];
        printf("Consuming item %d\n", item);
        pthread_mutex_unlock(&buffer_mutex);

        // Signal that an empty slot is available in the buffer
        sem_post(&empty_slots);

        // Simulate some work
        usleep(300000); // Sleep for 300 milliseconds
    }

    return NULL;
}

int main() {
    // Initialize counting semaphores
    sem_init(&empty_slots, 0, BUFFER_SIZE); // Number of empty slots in the buffer
}

```

```

sem_init(&filled_slots, 0, 0);      // Number of filled slots in the buffer

// Create producer and consumer threads
pthread_t producer_threads[NUM_PRODUCERS];
pthread_t consumer_threads[NUM_CONSUMERS];

for (int i = 0; i < NUM_PRODUCERS; ++i) {
    pthread_create(&producer_threads[i], NULL, producer, NULL);
}

for (int i = 0; i < NUM_CONSUMERS; ++i) {
    pthread_create(&consumer_threads[i], NULL, consumer, NULL);
}

// Wait for threads to finish (this won't happen in this example because the threads run indefinitely)
for (int i = 0; i < NUM_PRODUCERS; ++i) {
    pthread_join(producer_threads[i], NULL);
}

for (int i = 0; i < NUM_CONSUMERS; ++i) {
    pthread_join(consumer_threads[i], NULL);
}

// Clean up semaphores
sem_destroy(&empty_slots);
sem_destroy(&filled_slots);

return 0;
}

```

Dekker's Algorithm: Final and completed Solution – Idea is to use favoured thread notion to determine entry to the critical section. Favoured thread alternates between the thread providing mutual exclusion and avoiding deadlock, indefinite postponement, or lockstep synchronization.

- CPP

```

Main()
{
    // to denote which thread will enter next
    int favouredthread = 1;

    // flags to indicate if each thread is in
    // queue to enter its critical section
    boolean thread1wantstoenter = false;
    boolean thread2wantstoenter = false;

    startThreads();
}

```

```
Thread1()
{
    do {

        thread1wantstoenter = true;

        // entry section
        // wait until thread2 wants to enter
        // its critical section
        while (thread2wantstoenter == true) {

            // if 2nd thread is more favored
            if (favaouredthread == 2) {

                // gives access to other thread
                thread1wantstoenter = false;

                // wait until this thread is favored
                while (favouredthread == 2)
                    ;

                thread1wantstoenter = true;
            }
        }

        // critical section

        // favor the 2nd thread
        favouredthread = 2;

        // exit section
        // indicate thread1 has completed
        // its critical section
        thread1wantstoenter = false;

        // remainder section

    } while (completed == false)
}

Thread2()
{
```

```

do {

    thread2wantstoenter = true;

    // entry section
    // wait until thread1 wants to enter
    // its critical section
    while (thread1wantstoenter == true) {

        // if 1st thread is more favored
        if (favaouredthread == 1) {

            // gives access to other thread
            thread2wantstoenter = false;

            // wait until this thread is favored
            while (favaouredthread == 1)
                ;

            thread2wantstoenter = true;
        }
    }

    // critical section

    // favour the 1st thread
    favouredthread = 1;

    // exit section
    // indicate thread2 has completed
    // its critical section
    thread2wantstoenter = false;

    // remainder section

} while (completed == false)
}

```

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Bakery Algorithm is a critical section solution for N processes. The algorithm preserves the first come first serve property.

How does the Bakery Algorithm work?

In the Bakery Algorithm, each process is assigned a number (a ticket) in a lexicographical order. Before entering the critical section, a process receives a ticket number, and the

process with the smallest ticket number enters the critical section. If two processes receive the same ticket number, the process with the lower process ID is given priority.

How does the Bakery Algorithm ensure fairness?

The Bakery Algorithm ensures fairness by assigning a unique ticket number to each process based on a lexicographical order. This ensures that processes are served in the order they arrive, which guarantees that all processes will eventually enter the critical section.

- Before entering its critical section, the process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number,

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5, ...

Notation – lexicographical order (ticket #, process id #) – Firstly the ticket number is compared. If same then the process ID is compared next, i.e.-

Explanation – Firstly the process sets its “choosing” variable to be TRUE indicating its intent to enter critical section. Then it gets assigned the highest ticket number corresponding to other processes. Then the “choosing” variable is set to FALSE indicating that it now has a new ticket number. This is in-fact the most important and confusing part of the algorithm. It is actually a small critical section in itself ! The very purpose of the first three lines is that if a process is modifying its TICKET value then at that time some other process should not be allowed to check its old ticket value which is now obsolete. This is why inside the for loop before checking ticket value we first make sure that all other processes have the “choosing” variable as FALSE. After that we proceed to check the ticket values of processes where process with least ticket number/process id gets inside the critical section. The exit section just resets the ticket value to zero. **Code** – Here’s the C code implementation of the Bakery Algorithm. Run the following in a **UNIX environment** –

- C

```
// Importing the thread library
#include "pthread.h"

#include "stdio.h"

// Importing POSIX Operating System API library
#include "unistd.h"
```

```
#include "string.h"

// This is a memory barrier instruction.
// Causes compiler to enforce an ordering
// constraint on memory operations.
// This means that operations issued prior
// to the barrier will be performed
// before operations issued after the barrier.
#define MEMBAR __sync_synchronize()
#define THREAD_COUNT 8

volatile int tickets[THREAD_COUNT];
volatile int choosing[THREAD_COUNT];

// VOLATILE used to prevent the compiler
// from applying any optimizations.
volatile int resource;

void lock(int thread)
{
    // Before getting the ticket number
    // "choosing" variable is set to be true
    choosing[thread] = 1;

    MEMBAR;
    // Memory barrier applied

    int max_ticket = 0;

    // Finding Maximum ticket value among current threads
    for (int i = 0; i < THREAD_COUNT; ++i) {

        int ticket = tickets[i];
        max_ticket = ticket > max_ticket ? ticket : max_ticket;
    }

    // Allotting a new ticket value as MAXIMUM + 1
    tickets[thread] = max_ticket + 1;

    MEMBAR;
    choosing[thread] = 0;
    MEMBAR;
```

```
// The ENTRY Section starts from here
for (int other = 0; other < THREAD_COUNT; ++other) {

    // Applying the bakery algorithm conditions
    while (choosing[other]) {
    }

    MEMBAR;

    while (tickets[other] != 0 && (tickets[other]
                                    < tickets[thread]
                                    || (tickets[other]
                                        == tickets[thread]
                                        && other < thread))) {
        }

    }

// EXIT Section
void unlock(int thread)
{

    MEMBAR;
    tickets[thread] = 0;
}

// The CRITICAL Section
void use_resource(int thread)
{

    if (resource != 0) {
        printf("Resource was acquired by %d, but is still in-use by %d!\n",
               thread, resource);
    }

    resource = thread;
    printf("%d using resource...\n", thread);

    MEMBAR;
    sleep(2);
    resource = 0;
}
```

```

// A simplified function to show the implementation
void* thread_body(void* arg)
{
    long thread = (long)arg;
    lock(thread);
    use_resource(thread);
    unlock(thread);
    return NULL;
}

int main(int argc, char** argv)
{
    memset((void*)tickets, 0, sizeof(tickets));
    memset((void*)choosing, 0, sizeof(choosing));
    resource = 0;

    // Declaring the thread variables
    pthread_t threads[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; ++i) {

        // Creating a new thread with the function
        // "thread_body" as its thread routine
        pthread_create(&threads[i], NULL, &thread_body, (void*)((long)i));
    }

    for (int i = 0; i < THREAD_COUNT; ++i) {

        // Reaping the resources used by
        // all threads once their task is completed !
        pthread_join(threads[i], NULL);
    }

    return 0;
}

```

A [semaphore](#) S is an integer variable that can be accessed only through two standard operations : wait() and signal().
The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

Semaphores are of two types:

1. **Binary Semaphore** – This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore** – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section. To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Initialization of semaphores –

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Solution for Producer –

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

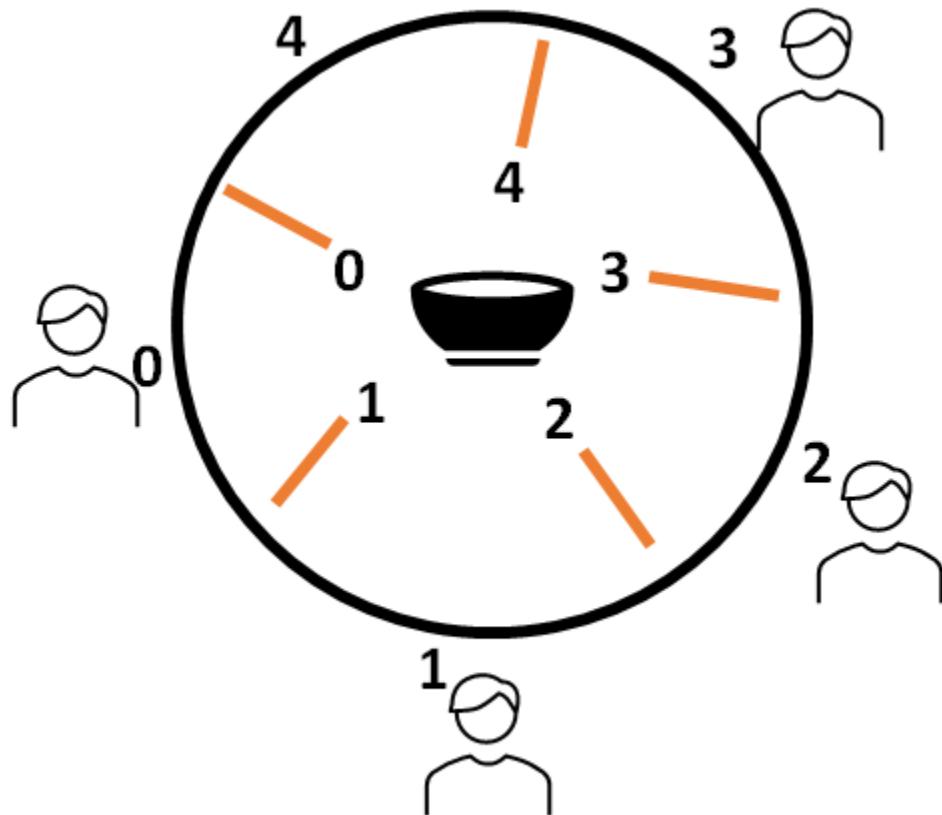
Solution for Consumer –

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value of mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

7. Program on Dining Philosopher Problem

[Programs](#) / By [Baljit Singh Saini](#)

In this post we are going to understand the C program implementing the solution to the Dining Philosopher Problem. The Dining Philosopher Problem states that there are five philosophers which do two thinks: think and eat. They share a table having a chair for each one of them. In the center of the table there is a bowl of rice and the table is laid with 5 single chopsticks (Refer Figure Below).



When a philosopher thinks, he does not interact with others. When he gets hungry, he tries to pick up the two chopsticks that are near to him. For example, philosopher 1 will try to pick chopsticks 1 and 2. But the philosopher can pickup only one chopstick at a time. He can not take a chopstick that is already in the hands of his neighbour. The philosopher stars to eat when he has both his chopsticks in his hand. After eating the philosopher puts down both the chopsticks and starts to think again.

Solution to Dining Philosopher Problem

Represent each chopstick with a semaphore. Each philosopher first picks up the left chopstick and then the right chopstick using the wait() operation each semaphore. After eating he puts down the chopsticks by using the signal() operation on each chopstick.

```
#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem_t chopstick[5];

void * philos(void *);

void eat(int);

int main()

{

    int i,n[5];

    pthread_t T[5];

    for(i=0;i<5;i++)

        sem_init(&chopstick[i],0,1);

    for(i=0;i<5;i++) {

        n[i]=i;

        pthread_create(&T[i],NULL,philos,(void *)&n[i]);

    }

    for(i=0;i<5;i++)

        pthread_join(T[i],NULL);

}
```

```
void * philos(void * n)

{
    int ph=*(int *)n;

    printf("Philosopher %d wants to eat\n",ph);

    printf("Philosopher %d tries to pick left chopstick\n",ph);

    sem_wait(&chopstick[ph]);

    printf("Philosopher %d picks the left chopstick\n",ph);

    printf("Philosopher %d tries to pick the right
chopstick\n",ph);

    sem_wait(&chopstick[(ph+1)%5]);

    printf("Philosopher %d picks the right chopstick\n",ph);

    eat(ph);

    sleep(2);

    printf("Philosopher %d has finished eating\n",ph);

    sem_post(&chopstick[(ph+1)%5]);

    printf("Philosopher %d leaves the right chopstick\n",ph);

    sem_post(&chopstick[ph]);

    printf("Philosopher %d leaves the left chopstick\n",ph);

}

void eat(int ph)

{
```

```
    printf("Philosopher %d begins to eat\n",ph);  
}
```

How it works?

Now let us understand how the code works

```
#include<stdio.h>  
  
#include<stdlib.h>  
  
#include<pthread.h>  
  
#include<semaphore.h>  
  
#include<unistd.h>  
  
sem_t chopstick[5];  
  
void * philos(void *);  
  
void eat(int);
```

`sem_t chopstick[5]` is used to declare 5 semaphore variable, one for each of the five chopsticks. Next, two are the prototypes for functions defined below.

In the `main()` function there are three for loops. The first loop is used to initialize each semaphore variable with initial value to 1. The second for loop is used to create five threads which will act as the five philosophers. The third for loop uses the `pthread_join` function which makes the parent program wait for each of the thread to finish.

Next is the `philos` function. The `philos` function when called by each thread receives the same value as the thread number. For example if thread one runs, the variable `ph` in the `philos` function is assigned the value `n`. This is done because each philosopher `n` before eating will pick two chopstick, `n` and $(n+1) \% 5$.

Next, the `sem_wait` function is used on the left chopstick first

```
sem_wait(&chopstick[ph]);
```

If successful, the thread executes the `sem_wait` function on the right chopstick

```
sem_wait(&chopstick[(ph+1)%5]);
```

These two operations are equivalent to picking the left chopstick and then the right chopstick. If both these operations are successful this means that the philosopher is able to pick both the chopsticks and hence will start to eat by calling the `eat()` function. After eating both the chopsticks are released by using the `sem_post()` function.

Output

Here's a sample output. Your output might differ each time you run the program. This is because the sequence of execution of threads will be different. Try to understand the output below and then relate it with what you get.

```
baljit@baljit:~/cse325/Synchronization$ ./a.out
Philosopher 0 wants to eat
Philosopher 0 tries to pick left chopstick
Philosopher 0 picks the left chopstick
Philosopher 0 tries to pick the right chopstick
Philosopher 0 picks the right chopstick
Philosopher 0 begins to eat
Philosopher 1 wants to eat
Philosopher 1 tries to pick left chopstick
Philosopher 3 wants to eat
Philosopher 3 tries to pick left chopstick
Philosopher 3 picks the left chopstick
Philosopher 3 tries to pick the right chopstick
Philosopher 3 picks the right chopstick
Philosopher 3 begins to eat
Philosopher 2 wants to eat
Philosopher 2 tries to pick left chopstick
Philosopher 2 picks the left chopstick
Philosopher 2 tries to pick the right chopstick
Philosopher 4 wants to eat
Philosopher 4 tries to pick left chopstick
Philosopher 0 has finished eating
Philosopher 0 leaves the right chopstick
Philosopher 0 leaves the left chopstick
Philosopher 1 picks the left chopstick
Philosopher 1 tries to pick the right chopstick
```

Here philosopher(thread) 0 tries to eat first. So, it tries to pick the left chopstick, which it does. Then the right one. Since it picks both the chopstick so philosopher 0 starts to eat. Now, refer to the image at the beginning of the post. If philosopher 0 starts to eat this means chopstick 0 and 1 are busy hence, philosopher 1 and 4 can not eat until philosopher 0 puts down the chopsticks. Read the output now, next philosopher wants to eat. It tries to pick the left chopstick (i.e. chopstick 1) but is not successful because chopstick 1 is already with philosopher 0. Similarly, you can understand the rest of the output.

Certainly! Let's go through the logic of the provided C code for the reader-writer problem. The example demonstrates a solution using mutex locks and semaphores to synchronize multiple reader and writer threads accessing a shared resource.

```
cCopy code
#include <stdio.h> #include <pthread.h> #include <semaphore.h> #include <unistd.h> #define
NUM_READERS 3 #define NUM_WRITERS 2 pthread_mutex_t mutex; // Mutex to protect critical sections
sem_t reader_semaphore; // Semaphore for controlling reader access sem_t writer_semaphore; // Semaphore
for controlling writer access int active_readers = 0; // Count of active readers void* reader(void* arg) {
while (1) { sem_wait(&reader_semaphore); pthread_mutex_lock(&mutex); active_readers++; if (active_readers == 1)
{ sem_wait(&writer_semaphore); } pthread_mutex_unlock(&mutex); sem_post(&reader_semaphore); // Critical
section: Reading printf("Reader %d is reading\n", *(int*)arg); usleep(200000); // Simulate reading time
sem_wait(&reader_semaphore); pthread_mutex_lock(&mutex); active_readers--; if (active_readers == 0) {
sem_post(&writer_semaphore); } pthread_mutex_unlock(&mutex); sem_post(&reader_semaphore); // Non-
critical section printf("Reader %d finished reading\n", *(int*)arg); // Sleep before the next iteration
usleep(300000); } return NULL; } void* writer(void* arg) { while (1) { sem_wait(&writer_semaphore); //
Critical section: Writing printf("Writer %d is writing\n", *(int*)arg); usleep(300000); // Simulate writing
time sem_post(&writer_semaphore); // Non-critical section printf("Writer %d finished writing\n",
*(int*)arg); // Sleep before the next iteration usleep(500000); } return NULL; } int main() { // Initialize
mutex and semaphores pthread_mutex_init(&mutex, NULL); sem_init(&reader_semaphore, 0, 1); // Binary
semaphore for reader access control sem_init(&writer_semaphore, 0, 1); // Binary semaphore for writer access
control // Create reader threads pthread_t reader_threads[NUM_READERS]; int reader_ids[NUM_READERS];
for (int i = 0; i < NUM_READERS; ++i) { reader_ids[i] = i; pthread_create(&reader_threads[i], NULL, reader,
&reader_ids[i]); } // Create writer threads pthread_t writer_threads[NUM_WRITERS]; int
writer_ids[NUM_WRITERS]; for (int i = 0; i < NUM_WRITERS; ++i) { writer_ids[i] = i;
pthread_create(&writer_threads[i], NULL, writer, &writer_ids[i]); } // Wait for threads to finish (this won't
happen in this example) for (int i = 0; i < NUM_READERS; ++i) { pthread_join(reader_threads[i], NULL); } for
(int i = 0; i < NUM_WRITERS; ++i) { pthread_join(writer_threads[i], NULL); } // Clean up
pthread_mutex_destroy(&mutex); sem_destroy(&reader_semaphore); sem_destroy(&writer_semaphore); return
0; }
```

Now, let's break down the logic step by step:

1. Initialization:

- Mutex (`mutex`), and semaphores (`reader_semaphore` and `writer_semaphore`) are initialized.
- `reader_semaphore` is a binary semaphore initially set to 1, allowing one reader at a time.
- `writer_semaphore` is a binary semaphore initially set to 1, allowing one writer at a time.

2. Reader Routine (reader function):

- Infinite loop (`while (1)`) simulates continuous operation of a reader.
- Acquire `reader_semaphore` to control access to the critical section for readers.
- Lock `mutex` to protect the `active_readers` counter.
- Increment `active_readers` and, if it's the first reader, wait on `writer_semaphore` to prevent writers from entering.
- Release `mutex` and `reader_semaphore`.
- Critical Section: Simulate reading with `printf` and `usleep`.

- Acquire `reader_semaphore` and `mutex` again.
- Decrement `active_readers` and, if it's the last reader, signal `writer_semaphore` to allow potential writers to enter.
- Release `mutex` and `reader_semaphore`.
- Non-critical Section: Print a message and sleep before the next iteration.

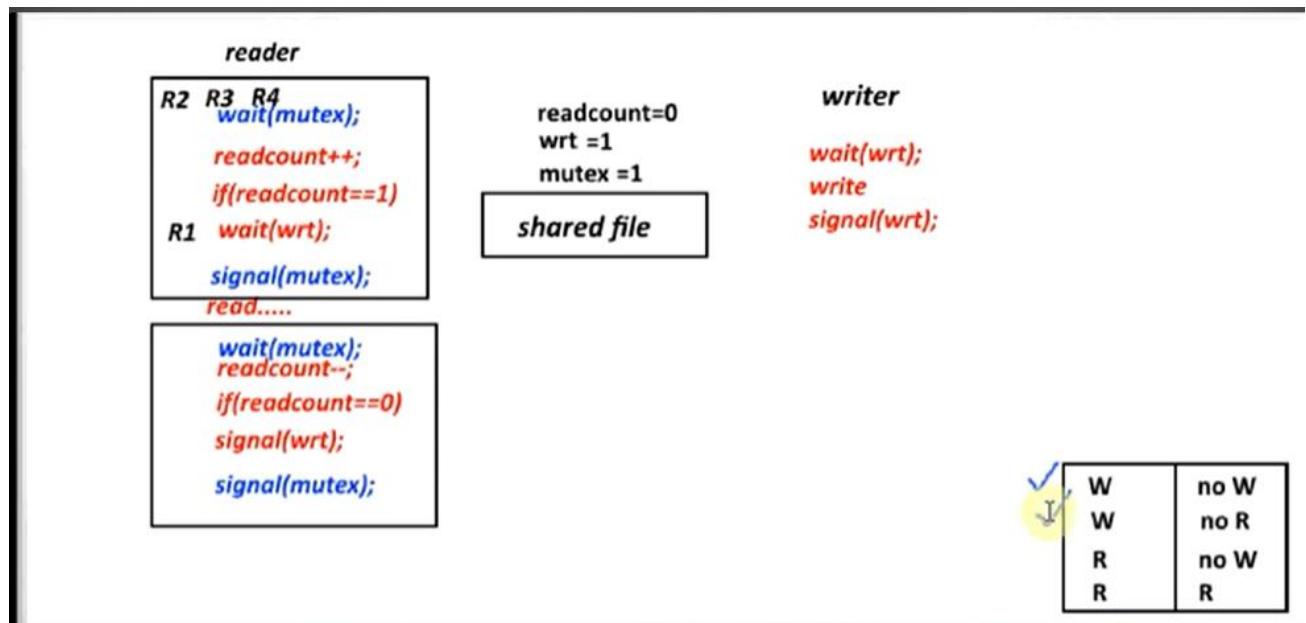
3. Writer Routine (writer function):

- Infinite loop (`while (1)`) simulates continuous operation of a writer.
- Acquire `writer_semaphore` to control access to the critical section for writers.
- Critical Section: Simulate writing with `printf` and `usleep`.
- Release `writer_semaphore`.
- Non-critical Section: Print a message and sleep before the next iteration.

4. Main Function:

- Create multiple reader and writer threads using `pthread_create`.
- Wait for threads to finish (this won't happen in this example as the threads run indefinitely).
- Clean up by destroying mutex and semaphores.

The provided code illustrates a solution to the reader-writer problem using semaphores and mutex locks to ensure proper synchronization and coordination between multiple reader and writer threads accessing a shared resource. The use of semaphores helps control access to critical sections and avoids conflicts between readers and writers. The simulated reading and writing times using `usleep` demonstrate the interleaved behavior of readers and



The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system. The problem is as follows:

There is a barber shop with one barber and a number of chairs for waiting customers. Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available. If there are no chairs available, the customer leaves. When the barber finishes with a customer, he checks if there are any waiting customers. If there are, he begins cutting the hair of the next customer in the queue. If there are no customers waiting, he goes to sleep.

The problem is to write a program that coordinates the actions of the customers and the barber in a way that avoids synchronization problems, such as deadlock or starvation.

One solution to the Sleeping Barber problem is to use semaphores to coordinate access to the waiting chairs and the barber chair. The solution involves the following steps:

Initialize two semaphores: one for the number of waiting chairs and one for the barber chair. The waiting chairs semaphore is initialized to the number of chairs, and the barber chair semaphore is initialized to zero.

Customers should acquire the waiting chairs semaphore before taking a seat in the waiting room. If there are no available chairs, they should leave.

When the barber finishes cutting a customer's hair, he releases the barber chair semaphore and checks if there are any waiting customers. If there are, he acquires the barber chair semaphore and begins cutting the hair of the next customer in the queue.

Learn more

[Pause](#)

[Unmute](#)

The barber should wait on the barber chair semaphore if there are no customers waiting.

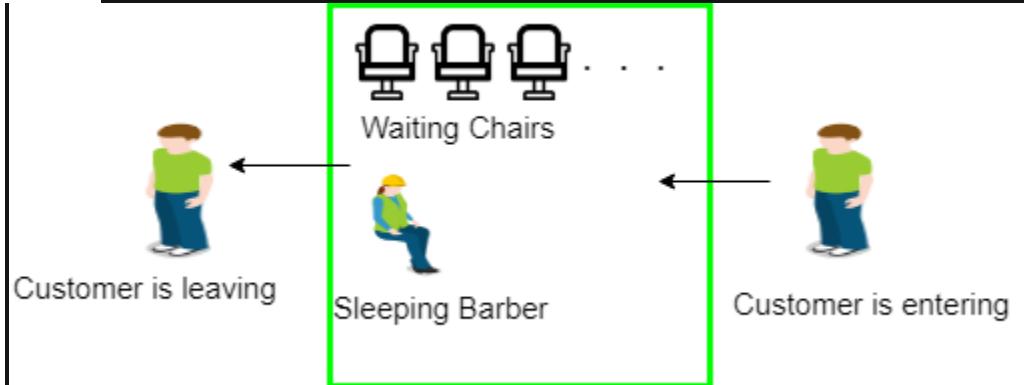
The solution ensures that the barber never cuts the hair of more than one customer at a time, and that customers wait if the barber is busy. It also ensures that the barber goes to sleep if there are no customers waiting.

However, there are variations of the problem that can require more complex synchronization mechanisms to avoid synchronization issues. For example, if multiple barbers are employed, a more complex mechanism may be needed to ensure that they do not interfere with each other.

Prerequisite – [Inter Process Communication Problem](#) : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

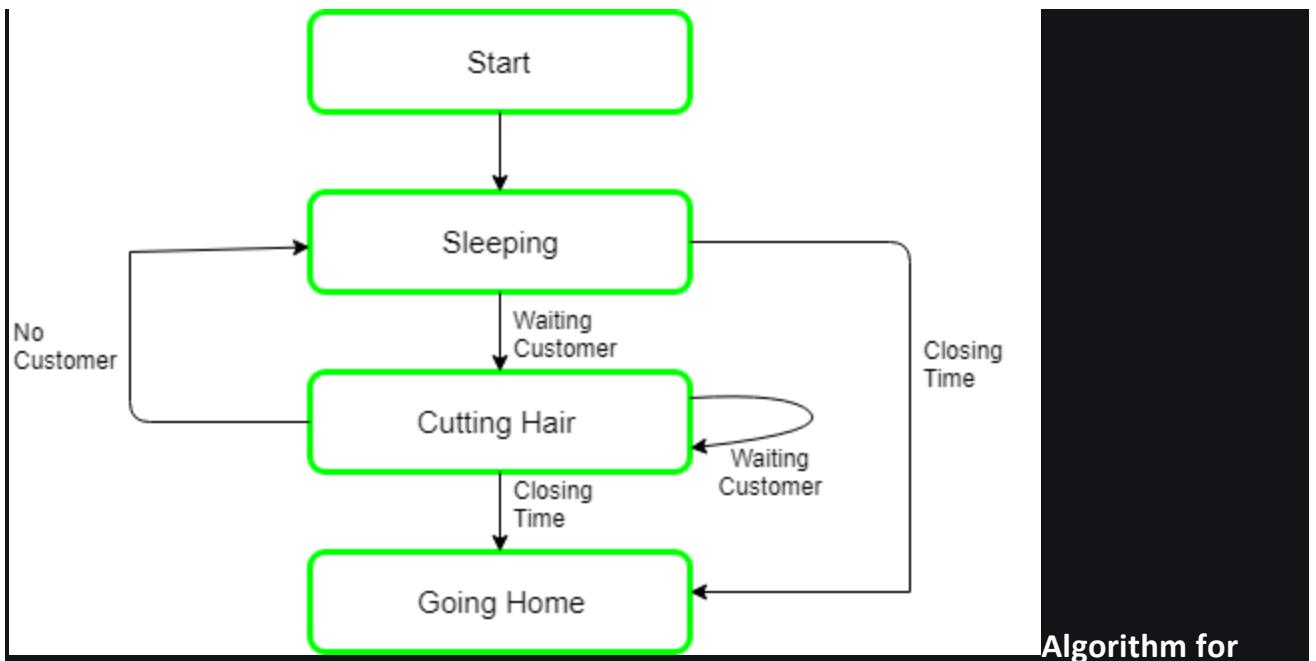
- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.

- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



Solution :

The solution to this problem includes three [semaphores](#). First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop. When the barber shows up in the morning, he executes the procedure `barber`, causing him to block on the semaphore `customers` because it is initially 0. Then the barber goes to sleep until the first customer comes up. When a customer arrives, he executes `customer` procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less than the number of chairs then he sits otherwise he leaves and releases the mutex. If the chair is available then customer sits in the waiting room and increments the variable `waiting` value and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



Algorithm for

Sleeping Barber problem:

- C

```

Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;

Barber {
    while(true) {
        /* waits for a customer (sleeps). */
        down(Customers);

        /* mutex to protect the number of available seats.*/
        down(Seats);

        /* a chair gets free.*/
        FreeSeats++;

        /* bring customer for haircut.*/
        up(Barber);

        /* release the mutex on the chair.*/
        up(Seats);
        /* barber is cutting hair.*/
    }
}
    
```

```

Customer {
    while(true) {
        /* protects seats so only 1 customer tries to sit
           in a chair if that's the case.*/
        down(Seats); //This line should not be here.
        if(FreeSeats > 0) {

            /* sitting down.*/
            FreeSeats--;

            /* notify the barber. */
            up(Customers);

            /* release the lock */
            up(Seats);

            /* wait in the waiting room if barber is busy. */
            down(Barber);
            // customer is having hair cut
        } else {
            /* release the lock */
            up(Seats);
            // customer leaves
        }
    }
}

```

A mutex is initialized and then a lock is achieved by calling the following two functions : The first function initializes a mutex and through second function any critical region in the code can be locked.

1. **int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr)** : Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

Returned value

If successful, `pthread_mutex_init()` returns 0, and the state of the mutex becomes initialized and unlocked.

If unsuccessful, `pthread_mutex_init()` returns -1.

2. **int pthread_mutex_lock(pthread_mutex_t *mutex)** : Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has

unlocked it. When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call `pthread_mutex_unlock()` the same number of times to decrement the count to zero.

Returned value

If successful, `pthread_mutex_lock()` returns 0.

If unsuccessful, `pthread_mutex_lock()` returns -1.

The mutex can be unlocked and destroyed by calling following two functions : The first function releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

1. **int `pthread_mutex_unlock(pthread_mutex_t *mutex)`** : Releases a mutex object. If one or more threads are waiting to lock the mutex, `pthread_mutex_unlock()` causes one of those threads to return from `pthread_mutex_lock()` with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner. When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting for it is posted.

Returned value

If successful, `pthread_mutex_unlock()` returns 0.

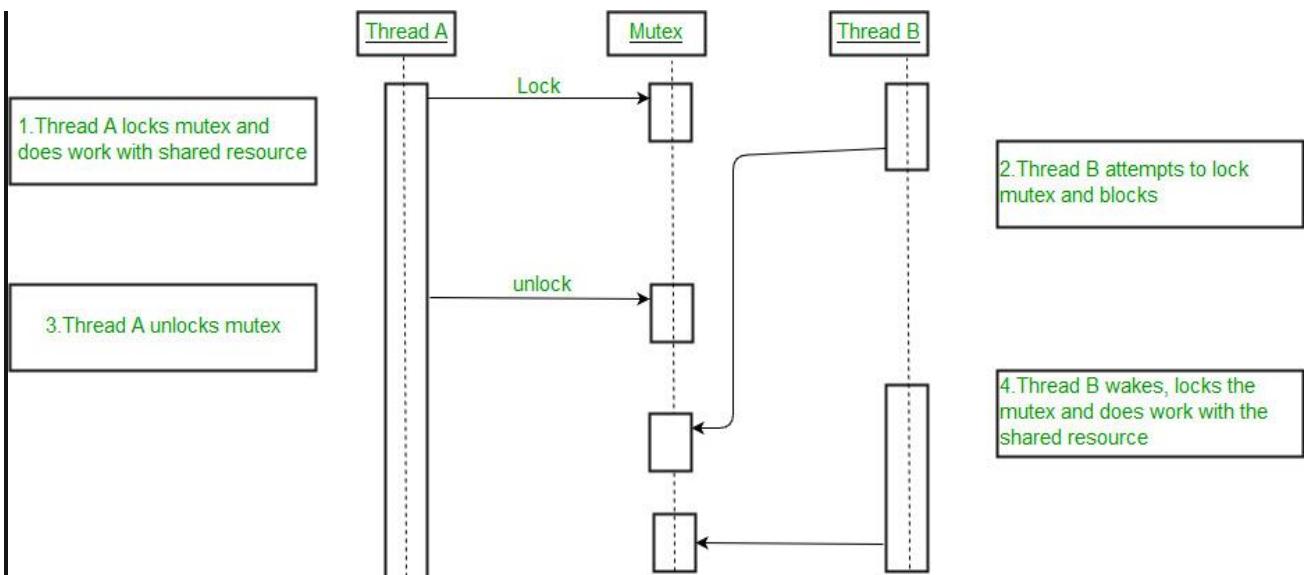
If unsuccessful, `pthread_mutex_unlock()` returns -1

2. **int `pthread_mutex_destroy(pthread_mutex_t *mutex)`** : Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using `pthread_mutex_init()`.

Returned value

If successful, `pthread_mutex_destroy()` returns 0.

If unsuccessful, `pthread_mutex_destroy()` returns -1.



An example to show how mutexes are used for thread synchronization

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```

int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

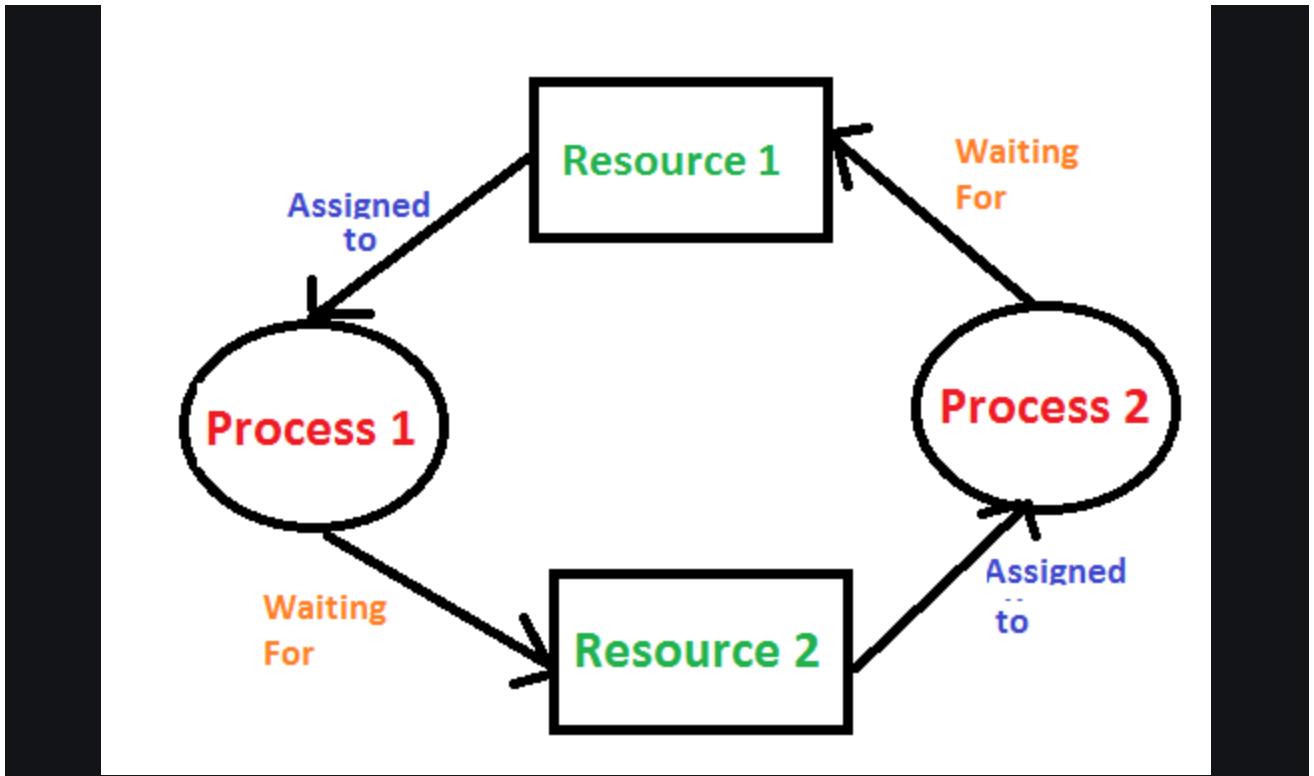
    return 0;
}

```

A process in operating system uses resources in the following way.

1. Requests a resource
2. Use the resource
3. Releases the resource

A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Examples Of Deadlock

1. The system has 2 tape drives. P1 and P2 each hold one tape drive and each needs another one.
2. Semaphores A and B, initialized to 1, P0, and P1 are in deadlock as follows:
 - P0 executes wait(A) and preempts.
 - P1 executes wait(B).
 - Now P0 and P1 enter in deadlock.

P0	P1
wait(A);	wait(B)
wait(B);	wait(A)

3. Assume the space is available for allocation of 200K bytes, and the following sequence of events occurs.

P0	P1
Request 80KB;	Request 70KB;

P0	P1
Request 60KB;	Request 80KB;
Deadlock occurs if both processes progress to their second request.	
Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)	
Mutual Exclusion: Two or more resources are non-shareable (Only one process can use at a time)	
Hold and Wait: A process is holding at least one resource and waiting for resources.	
No Preemption: A resource cannot be taken from a process unless the process releases the resource.	
Circular Wait: A set of processes waiting for each other in circular form.	
Methods for handling deadlock	
There are three ways to handle deadlock	
1) Deadlock prevention or avoidance:	
Prevention:	
The idea is to not let the system into a deadlock state. This system will make sure that above mentioned four conditions will not arise. These techniques are very costly so we use this in cases where our priority is making a system deadlock-free.	
One can zoom into each category individually, Prevention is done by negating one of the above-mentioned necessary conditions for deadlock. Prevention can be done in four different ways:	
<ul style="list-style-type: none"> 1. Eliminate mutual exclusion 2. Solve hold and Wait 3. Allow preemption 4. Circular wait Solution 	
Avoidance:	
Avoidance is kind of futuristic. By using the strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process. We use Banker's algorithm (Which is in turn a gift from Dijkstra) to avoid deadlock.	
In prevention and avoidance, we get the correctness of data but performance decreases.	
2) Deadlock detection and recovery: If Deadlock prevention or avoidance is not applied to the software then we can handle this by deadlock detection and recovery. which consist of two phases:	
<ul style="list-style-type: none"> 1. In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system. 2. If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock. 	
In Deadlock detection and recovery, we get the correctness of data but performance decreases.	
Recovery from Deadlock	

1. Manual Intervention:

When a deadlock is detected, one option is to inform the operator and let them handle the situation manually. While this approach allows for human judgment and decision-making, it can be time-consuming and may not be feasible in large-scale systems.

2. Automatic Recovery:

An alternative approach is to enable the system to recover from deadlock automatically. This method involves breaking the deadlock cycle by either aborting processes or preempting resources. Let's delve into these strategies in more detail.

Recovery from Deadlock: **Process Termination**:

1. Abort all deadlocked processes:

This approach breaks the deadlock cycle, but it comes at a significant cost. The processes that were aborted may have executed for a considerable amount of time, resulting in the loss of partial computations. These computations may need to be recomputed later.

2. Abort one process at a time:

Instead of aborting all deadlocked processes simultaneously, this strategy involves selectively aborting one process at a time until the deadlock cycle is eliminated. However, this incurs overhead as a deadlock-detection algorithm must be invoked after each process termination to determine if any processes are still deadlocked.

Factors for choosing the termination order:

- The process's priority
- Completion time and the progress made so far
- Resources consumed by the process
- Resources required to complete the process
- Number of processes to be terminated
- Process type (interactive or batch)

Recovery from Deadlock: **Resource Preemption**:

1. Selecting a victim:

Resource preemption involves choosing which resources and processes should be preempted to break the deadlock. The selection order aims to minimize the overall cost of recovery. Factors considered for victim selection may include the number of resources held by a deadlocked process and the amount of time the process has consumed.

2. Rollback:

If a resource is preempted from a process, the process cannot continue its normal execution as it lacks the required resource. Rolling back the process to a safe state and restarting it is a common approach. Determining a safe state can be challenging, leading to the use of total rollback, where the process is aborted and restarted from scratch.

3. Starvation prevention:

To prevent resource starvation, it is essential to ensure that the same process is not always chosen as a victim. If victim selection is solely based on cost factors, one process might repeatedly lose its resources and never complete its designated task. To address this, it is advisable to limit the number of times a process can be chosen as a victim, including the number of rollbacks in the cost factor.

3) Deadlock ignorance: If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. we use the ostrich algorithm for deadlock ignorance.

In Deadlock, ignorance performance is better than the above two methods but the correctness of data.

Safe State:

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

When two or more processes try to access the critical section at the same time and they fail to access simultaneously or stuck while accessing the critical section then this condition is known as Deadlock.

1. Every process needs a few resources to finish running.
2. The procedure makes a resource request. If the resource is available, the OS will grant it; otherwise, the process will wait.
3. When the process is finished, it is released.

7.1 Deadlock Characteristics

The deadlock has the following characteristics:

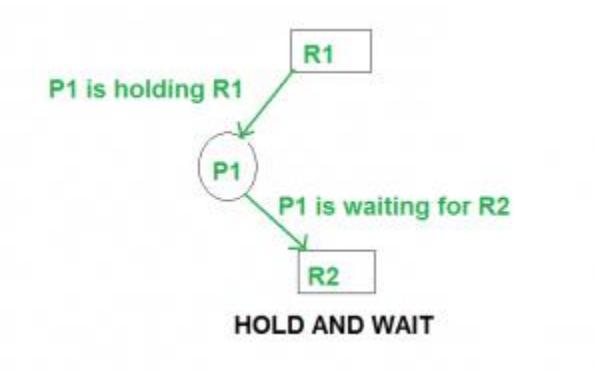
1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

7.2 Deadlock Prevention

We can prevent a Deadlock by eliminating any of the above four conditions.

Eliminate Mutual Exclusion: It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Eliminate Hold and wait: Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires a printer at a later time and we have allocated a printer before the start of its execution printer will remain blocked till it has completed its execution. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.



Eliminate No Preemption : Preempt resources from the process when resources are required by other high-priority processes.

Eliminate Circular Wait : Each resource will be assigned a numerical number. A process can request the resources to increase/decrease. order of numbering. For Example, if the P1 process is allocated R5 resources, now next time if P1 asks for R4, R3 lesser than R5 such a request will not be granted, only a request for resources more than R5 will be granted.

Detection and Recovery: Another approach to dealing with deadlocks is to detect and recover from them when they occur. This can involve killing one or more of the processes involved in the deadlock or releasing some of the resources they hold.

7.3 Deadlock Avoidance

A deadlock avoidance policy grants a resource request only if it can establish that granting the request cannot lead to a deadlock either immediately or in the future. The kernel lacks detailed knowledge about future behavior of processes, so it cannot accurately predict deadlocks. To facilitate deadlock avoidance under these conditions, it uses the following conservative approach: Each process declares the maximum number of resource units of each class that it may require. The kernel permits a process to request these resource units in stages- i.e. a few resource units at a time- subject to the maximum number declared by it and uses a worst case analysis technique to check for the possibility of future deadlocks. A request is granted only if there is no possibility of deadlocks; otherwise, it remains pending until it can be granted. This approach is conservative because a process may complete its operation without requiring the maximum number of units declared by it.

Resource Allocation Graph

The resource allocation graph (RAG) is used to visualize the system's current state as a graph. The Graph includes all processes, the resources that are assigned to them, as well as the resources that each Process requests. Sometimes, if there are fewer processes, we can quickly spot a deadlock in the system by looking at the graph rather than the tables we use in [Banker's algorithm](#). Deadlock avoidance can also be done with Banker's Algorithm.

Banker's Algorithm

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, and after granting a request system remains in the safe state it allows the request, and if there is no safe state it doesn't allow the request made by the process.

7.3.1.1 Inputs to Banker's Algorithm

1. Max needs of resources by each process.
2. Currently, allocated resources by each process.
3. Max free available resources in the system.

7.3.1.2 The request will only be granted under the below condition

1. If the request made by the process is less than equal to the max needed for that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Timeouts: To avoid deadlocks caused by indefinite waiting, a timeout mechanism can be used to limit the amount of time a process can wait for a resource. If the help is unavailable within the timeout period, the process can be forced to release its current resources and try again later.

Example:

The total number of resources are

Available resources are

Maximum resources we have for a process

Need = Maximum Resources Requirement – Currently Allocated Resources.

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system. A deadlock occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system-wide stall, where no process can make progress.

There are two main approaches to deadlock detection and recovery:

1. **Prevention:** The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

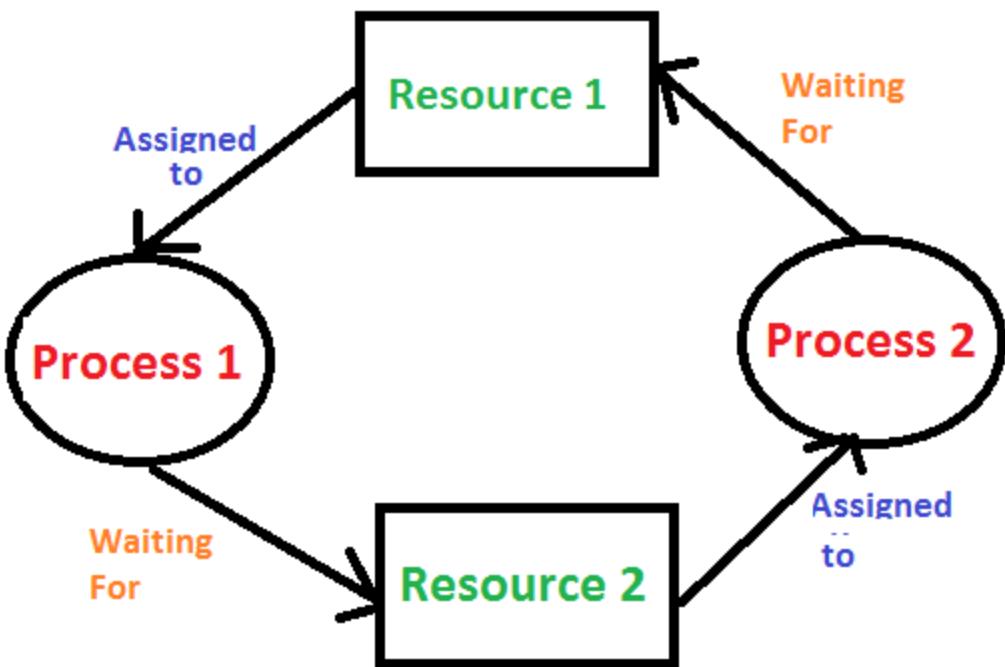
Difference Between Prevention and Detection/Recovery: Prevention aims to avoid deadlocks altogether by carefully managing resource allocation, while detection and recovery aim to identify and resolve deadlocks that have already occurred.

Deadlock detection and recovery is an important aspect of operating system design and management, as it affects the stability and performance of the system. The choice of deadlock detection and recovery approach depends on the specific requirements of the system and the trade-offs between performance, complexity, and risk tolerance. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

Deadlock Detection :

1. If resources have a single instance –

In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle $R_1 \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1$. So, Deadlock is Confirmed.

2. If there are multiple instances of resources –

Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

3. Wait-For Graph Algorithm –

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

Deadlock Recovery :

A traditional operating system such as Windows doesn't deal with deadlock recovery as it is a time and space-consuming process. Real-time operating systems use Deadlock recovery.

1. Killing the process –

Killing all the processes involved in the deadlock. Killing process one by one. After killing each process check for deadlock again and keep repeating the process till the system recovers from deadlock. Killing all the processes one by one helps a system to break circular wait conditions.

2. Resource Preemption –

Resources are preempted from the processes involved in the deadlock, and

preempted resources are allocated to other processes so that there is a possibility of recovering the system from the deadlock. In this case, the system goes into starvation.

3. **Concurrency Control** – Concurrency control mechanisms are used to prevent data inconsistencies in systems with multiple concurrent processes. These mechanisms ensure that concurrent processes do not access the same data at the same time, which can lead to inconsistencies and errors. Deadlocks can occur in concurrent systems when two or more processes are blocked, waiting for each other to release the resources they need. This can result in a system-wide stall, where no process can make progress. Concurrency control mechanisms can help prevent deadlocks by managing access to shared resources and ensuring that concurrent processes do not interfere with each other.

ADVANTAGES OR DISADVANTAGES:

Advantages of Deadlock Detection and Recovery in Operating Systems:

1. **Improved System Stability:** Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.
2. **Better Resource Utilization:** By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. **Better System Design:** Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.

Disadvantages of Deadlock Detection and Recovery in Operating Systems:

1. **Performance Overhead:** Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action to resolve them.
2. **Complexity:** Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.
3. **False Positives and Negatives:** Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
4. **Risk of Data Loss:** In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Overall, the choice of deadlock detection and recovery approach depends on the specific requirements of the system, the trade-offs between performance, complexity, and accuracy, and the risk tolerance of the system. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

Creating a specific livelock example in C can be a bit tricky because it often depends on the specific context of the program. However, I can provide a simplified illustration of a potential livelock scenario involving two threads using mutual exclusion. Please note that this is a contrived example, and livelocks in real-world scenarios are usually more complex.

[Copy code](#)

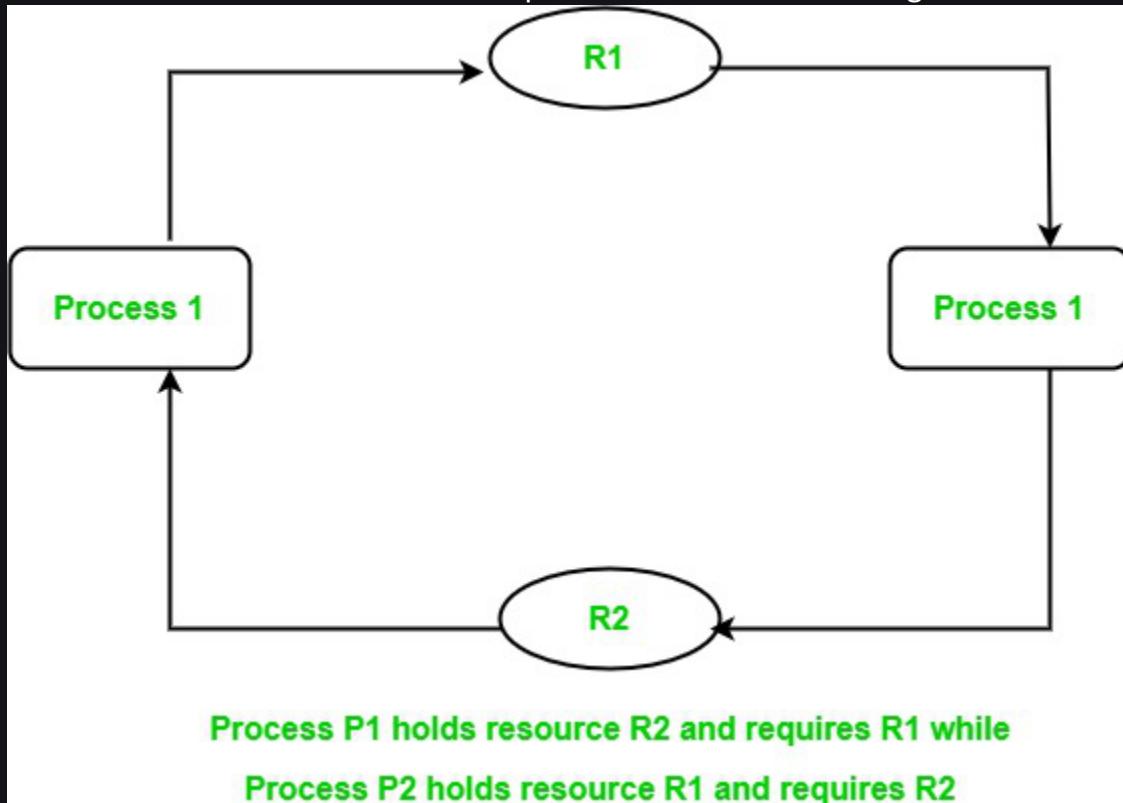
```
#include <stdio.h> #include <pthread.h> #include <unistd.h> pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER; pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER; void* threadA(void* arg) { while (1) { printf("Thread A trying to acquire mutex A...\n"); pthread_mutex_lock(&mutexA); // Simulate some work usleep(100000); printf("Thread A trying to acquire mutex B...\n"); if (pthread_mutex_trylock(&mutexB) == 0) { // Successfully acquired mutex B printf("Thread A acquired mutex B.\n"); pthread_mutex_unlock(&mutexB); } else { // Failed to acquire mutex B, release mutexA printf("Thread A failed to acquire mutex B. Releasing mutex A.\n"); pthread_mutex_unlock(&mutexA); } return NULL; } void* threadB(void* arg) { while (1) { printf("Thread B trying to acquire mutex B...\n"); pthread_mutex_lock(&mutexB); // Simulate some work usleep(100000); printf("Thread B trying to acquire mutex A...\n"); if (pthread_mutex_trylock(&mutexA) == 0) { // Successfully acquired mutex A printf("Thread B acquired mutex A.\n"); pthread_mutex_unlock(&mutexA); } else { // Failed to acquire mutex A, release mutexB printf("Thread B failed to acquire mutex A. Releasing mutex B.\n"); pthread_mutex_unlock(&mutexB); } return NULL; } int main() { pthread_t threadA_id, threadB_id; // Create threads pthread_create(&threadA_id, NULL, threadA, NULL); pthread_create(&threadB_id, NULL, threadB, NULL); // Wait for threads to finish (this won't happen in this example) pthread_join(threadA_id, NULL); pthread_join(threadB_id, NULL); return 0; }
```

In this example:

- Two threads, `threadA` and `threadB`, are each trying to acquire two mutexes (`mutexA` and `mutexB`).
- Both threads perform some work after acquiring the first mutex and then attempt to acquire the second mutex.
- The threads use `pthread_mutex_trylock` to attempt to acquire the second mutex without blocking. If unsuccessful, they release the first mutex.
- This setup creates a situation where both threads are repeatedly trying to acquire the second mutex, and neither can make progress because they keep releasing the first mutex when the other thread tries to acquire it.

Please note that this example is simplified, and creating a real-world livelock often involves more complex interactions and conditions. Detecting and resolving livelocks typically require careful analysis of the program's logic and synchronization mechanisms. In practice, it's essential to design concurrent programs with proper synchronization to avoid livelocks and ensure that threads can make progress.

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.



Example: Imagine a pair of processes using two resources, as shown:

- CPP

```
void process_A(void)
{
    enter_reg(& resource_1);
    enter_reg(& resource_2);
    use_both_resources();
    leave_reg(& resource_2);
    leave_reg(& resource_1);
}
void process_B(void)
{
    enter_reg(& resource_1);
    enter_reg(& resource_2);
    use_both_resources();
    leave_reg(& resource_2);
```

```
    leave_reg(& resource_1);
}
```

Each of the two processes needs the two resources and they use the polling primitive enter_reg to try to acquire the locks necessary for them. In case, the attempt fails, the process just tries again. If process A runs first and acquires resource 1 and then process B runs and acquires resource 2, no matter which one runs next, it will make no further progress, but neither of the two processes blocks. What actually happens is that it uses its CPU quantum over and over again without any progress being made but also without any sort of blocking. Thus, this situation is not that of a deadlock(as no process is being blocked) but we have something functionally equivalent to a deadlock: LIVELOCK.

What leads to Livelocks?

The occurrence of livelocks can occur in the most surprising ways. The total number of allowed processes in some systems is determined by the number of entries in the process table. Thus process table slots can be referred to as Finite Resources. If a fork fails because of the table being full, waiting a random time and trying again would be a reasonable approach for the program doing the fork. Consider a UNIX system having 100 process slots. Ten programs are running, each of which has to create 12 (sub)processes. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing – which is aptly the situation of a deadlock. The probability of this happening is very little but it could happen.

Difference between Deadlock, Starvation, and Livelock:

A **livelock** is similar to a **deadlock**, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource **starvation**; the general definition states that a specific process is not progressing.

Deadlock:

- CPP

```
var p = new object();
lock(p)
{
    lock(p)
    {
        // deadlock. Since p is previously locked
        // we will never reach here...
```

```
}
```

A *deadlock* is a state in which each member of a group of actions, is waiting for some other member to release a lock. A *livelock* on the other hand is almost similar to a deadlock, except that the states of the processes involved in a livelock constantly keep on changing with regard to one another, none progressing. Thus Livelock is a special case of resource starvation, as stated from the general definition, the process is not progressing.

Starvation:

Starvation is a problem that is closely related to both, Livelock and Deadlock. In a dynamic system, requests for resources keep on happening. Thereby, some policy is needed to make a decision about who gets the resource and when this process, being reasonable, may lead to some processes never getting serviced even though they are not deadlocked.

- CPP

```
Queue q = .....

while (q.Count > 0)
{
    var c = q.Dequeue();
    .....

    // Some method in different thread accidentally
    // puts c back in queue twice within same time frame
    q.Enqueue(c);
    q.Enqueue(c);

    // leading to growth of queue twice then it
    // can consume, thus starving of computing
}
```

Starvation happens when “greedy” threads make shared resources unavailable for long periods. For instance, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Starvation in concurrent programming occurs when a thread or process is unable to gain necessary resources to make progress. This typically happens when a resource is contended, and other threads or processes consistently acquire and release the resource, preventing the starving thread from gaining access. Starvation can lead to a situation where certain threads are neglected and unable to execute, even though they are ready.

Let's illustrate starvation with a simple example in C using pthreads and a shared resource protected by a mutex. In this example, multiple threads contend for access to a critical section, and one of the threads might experience starvation.

```
cCopy code
#include <stdio.h> #include <pthread.h> #include <unistd.h> #define NUM_THREADS 5
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* worker(void* arg) {
    int thread_id = *(int*)arg;
    while (1) {
        // Acquire the mutex to enter the critical section
        pthread_mutex_lock(&mutex); // Critical section: Simulate some work
        printf("Thread %d is in the critical section\n", thread_id);
        usleep(200000); // Simulate work
        // Release the mutex
        pthread_mutex_unlock(&mutex); // Non-critical section: Simulate some additional work
        printf("Thread %d is in the non-critical section\n", thread_id);
        usleep(300000); // Simulate additional work
    }
}
return NULL;
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS]; // Create threads
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, worker, &thread_ids[i]);
    } // Wait for threads to finish (this won't happen in this example)
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

In this example:

- Multiple threads (workers) are created, and each thread executes a simple loop.
- Each thread tries to enter a critical section protected by a mutex (`pthread_mutex_t`).
- After entering the critical section, the thread simulates some work and then releases the mutex.
- The thread then performs some non-critical section work.

Now, let's discuss the potential for starvation:

- If one or more threads frequently acquire and release the mutex, other threads may be starved because they struggle to gain access to the critical section.
- Starvation can occur if certain threads are consistently unable to acquire the mutex due to competition from other threads.

This example is intentionally simplified to highlight the concept of starvation. In real-world scenarios, starvation can be more subtle and may involve multiple factors, such as priority inversion, priority scheduling, and the overall design of the concurrent system. Detecting and addressing starvation often requires careful analysis of the system's design and the scheduling policies in use.

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

7.4 Why Banker's Algorithm is Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of

account holders in a bank and the total sum of their money is S . If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future. In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.

The following **Data structures** are used to implement the Banker's Algorithm:

Let ' n ' be the number of processes in the system and ' m ' be the number of resource types.

Available

- It is a 1-d array of size ' m ' indicating the number of available resources of each type.
- $\text{Available}[j] = k$ means there are ' k ' instances of resource type R_j

Max

- It is a 2-d array of size ' $n*m$ ' that defines the maximum demand of each process in a system.
- $\text{Max}[i, j] = k$ means process P_i may request at most ' k ' instances of resource type R_j .

Allocation

- It is a 2-d array of size ' $n*m$ ' that defines the number of resources of each type currently allocated to each process.
- $\text{Allocation}[i, j] = k$ means process P_i is currently allocated ' k ' instances of resource type R_j

Need

- It is a 2-d array of size ' $n*m$ ' that indicates the remaining resource need of each process.
- $\text{Need} [i, j] = k$ means process P_i currently needs ' k ' instances of resource type R_j
- $\text{Need} [i, j] = \text{Max} [i, j] - \text{Allocation} [i, j]$

Allocation specifies the resources currently allocated to process P_i and Need $_i$ specifies the additional resources that process P_i may still request to complete its task. Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

Banker's Algorithm

1. **Active**:= Running U Blocked;
for $k=1...r$

```

    New_request[k]:= Requested_resources[requesting_process, k];
2. Simulated_allocation:= Allocated_resources;
   for k=1.....r //Compute projected allocation state
      Simulated_allocation [requesting_process, k]:= Simulated_allocation [requesting
      _process, k] + New_request[k];
3. feasible:= true;
   for k=1....r // Check whether projected allocation state is feasible
   if Total_resources[k]< Simulated_total_alloc [k] then feasible:= false;
4. if feasible= true
   then // Check whether projected allocation state is a safe allocation state
   while set Active contains a process P1 such that
      For all k, Total_resources[k] – Simulated_total_alloc[k] >= Max_need [l,k]-Simulated_
      allocation[l, k]
      Delete P1 from Active;
   for k=1....r
      Simulated_total_alloc[k]:= Simulated_total_alloc[k]- Simulated_allocation[l, k];
5. If set Active is empty
   then // Projected allocation state is a safe allocation state
   for k=1....r // Delete the request from pending requests
      Requested_resources[requesting_process, k]:=0;
   for k=1....r // Grant the request
      Allocated_resources[requesting_process, k]:= Allocated_resources[requesting_
      process, k] + New_request[k];
      Total_alloc[k]:= Total_alloc[k] + New_request[k];

```

7.5 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

Resource-Request

Let Request_i be the request array for process P_i. Request_i [j] = k means process P_i wants k

Algorithm

instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time to following snapshot of the system has been taken:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Q.1: What will be the content of the Need matrix?

$$\text{Need } [i, j] = \text{Max } [i, j] - \text{Allocation } [i, j]$$

So, the content of Need Matrix is:

Process	Need		
	A	B	C
P ₀	7	4	3
P ₁	1	2	2
P ₂	6	0	0
P ₃	0	1	1
P ₄	4	3	1

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?
Applying the Safety algorithm on the given system,

m=3, n=5 Step 1 of Safety Algo Work = Available Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>3</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>false</td><td>false</td><td>false</td><td>false</td></tr></table>	3	3	2	0	1	2	3	4	false	false	false	false	false	For i=3 Need ₃ = 0, 1, 1 Finish [3] = false and Need ₃ < Work So P ₃ must be kept in safe sequence Step 2: Work = Work + Allocation ₃ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>3</td><td>2</td><td>1</td><td>1</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>true</td><td>true</td><td>false</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	5	3	2	1	1	true	true	false	true	true	7, 4, 5 0, 1, 0 Step 3 Work = Work + Allocation ₀ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>5</td><td>5</td><td>0</td><td>1</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>true</td><td>true</td><td>false</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	7	5	5	0	1	true	true	false	true	true
3	3	2																																											
0	1	2	3	4																																									
false	false	false	false	false																																									
0	1	2	3	4																																									
5	3	2	1	1																																									
true	true	false	true	true																																									
0	1	2	3	4																																									
7	5	5	0	1																																									
true	true	false	true	true																																									
For i=0 Need ₀ = 7, 4, 3 Finish [0] is false and Need ₀ > Work So P ₀ must wait But Need ≤ Work	Step 2: Work = Work + Allocation ₃ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>4</td><td>3</td><td>2</td><td>1</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>true</td><td>false</td></tr></table>	0	1	2	3	4	7	4	3	2	1	false	true	false	true	false	5, 3, 2 2, 1, 1 Step 3 Work = Work + Allocation ₃ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>4</td><td>3</td><td>2</td><td>1</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>true</td><td>false</td></tr></table>	0	1	2	3	4	7	4	3	2	1	false	true	false	true	false													
0	1	2	3	4																																									
7	4	3	2	1																																									
false	true	false	true	false																																									
0	1	2	3	4																																									
7	4	3	2	1																																									
false	true	false	true	false																																									
For i=1 Need ₁ = 1, 2, 2 Finish [1] is false and Need ₁ < Work So P ₁ must be kept in safe sequence	Step 2: Work = Work + Allocation ₁ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>3</td><td>2</td><td>1</td><td>2</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>false</td><td>false</td></tr></table>	0	1	2	3	4	5	3	2	1	2	false	true	false	false	false	4, 3, 1 7, 4, 3 Step 2: Work = Work + Allocation ₂ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>10</td><td>5</td><td>7</td><td>6</td><td>0</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>true</td><td>true</td><td>true</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	10	5	7	6	0	true	true	true	true	true													
0	1	2	3	4																																									
5	3	2	1	2																																									
false	true	false	false	false																																									
0	1	2	3	4																																									
10	5	7	6	0																																									
true	true	true	true	true																																									
3, 3, 2 2, 0, 0 Work = Work + Allocation ₁ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>3</td><td>2</td><td>1</td><td>2</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>false</td><td>false</td></tr></table>	0	1	2	3	4	5	3	2	1	2	false	true	false	false	false	For i=4 Need ₄ = 4, 3, 1 Finish [4] = false and Need ₄ < Work So P ₄ must be kept in safe sequence	7, 5, 5 3, 0, 2 Step 3 Work = Work + Allocation ₂ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>10</td><td>5</td><td>7</td><td>6</td><td>0</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>true</td><td>true</td><td>true</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	10	5	7	6	0	true	true	true	true	true													
0	1	2	3	4																																									
5	3	2	1	2																																									
false	true	false	false	false																																									
0	1	2	3	4																																									
10	5	7	6	0																																									
true	true	true	true	true																																									
For i=2 Need ₂ = 6, 0, 0 Finish [2] is false and Need ₂ > Work So P ₂ must wait	Step 2: Work = Work + Allocation ₄ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>4</td><td>5</td><td>6</td><td>0</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	7	4	5	6	0	false	true	false	true	true	7, 4, 3 0, 0, 2 Step 3 Work = Work + Allocation ₄ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>4</td><td>5</td><td>6</td><td>0</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	7	4	5	6	0	false	true	false	true	true													
0	1	2	3	4																																									
7	4	5	6	0																																									
false	true	false	true	true																																									
0	1	2	3	4																																									
7	4	5	6	0																																									
false	true	false	true	true																																									
For i=0 Need ₀ = 7, 4, 3 Finish [0] is false and Need ₀ < Work So P ₀ must be kept in safe sequence	Step 2: Work = Work + Allocation ₀ Work = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>7</td><td>4</td><td>3</td><td>6</td><td>0</td></tr></table> Finish = <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr></table>	0	1	2	3	4	7	4	3	6	0	false	true	false	true	true	Finish [i] = true for 0 ≤ i ≤ n Hence the system is in Safe state Step 4																												
0	1	2	3	4																																									
7	4	3	6	0																																									
false	true	false	true	true																																									

The safe sequence is P₁, P₃, P₄, P₀, P₂

Q.3: What will happen if process P₁ requests one additional instance of resource type A and two instances of resource type C?

A B C
Request₁ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1
1, 0, 2 1, 2, 2
Request₁ < Need₁

Step 2
1, 0, 2 3, 3, 2
Request₁ < Available

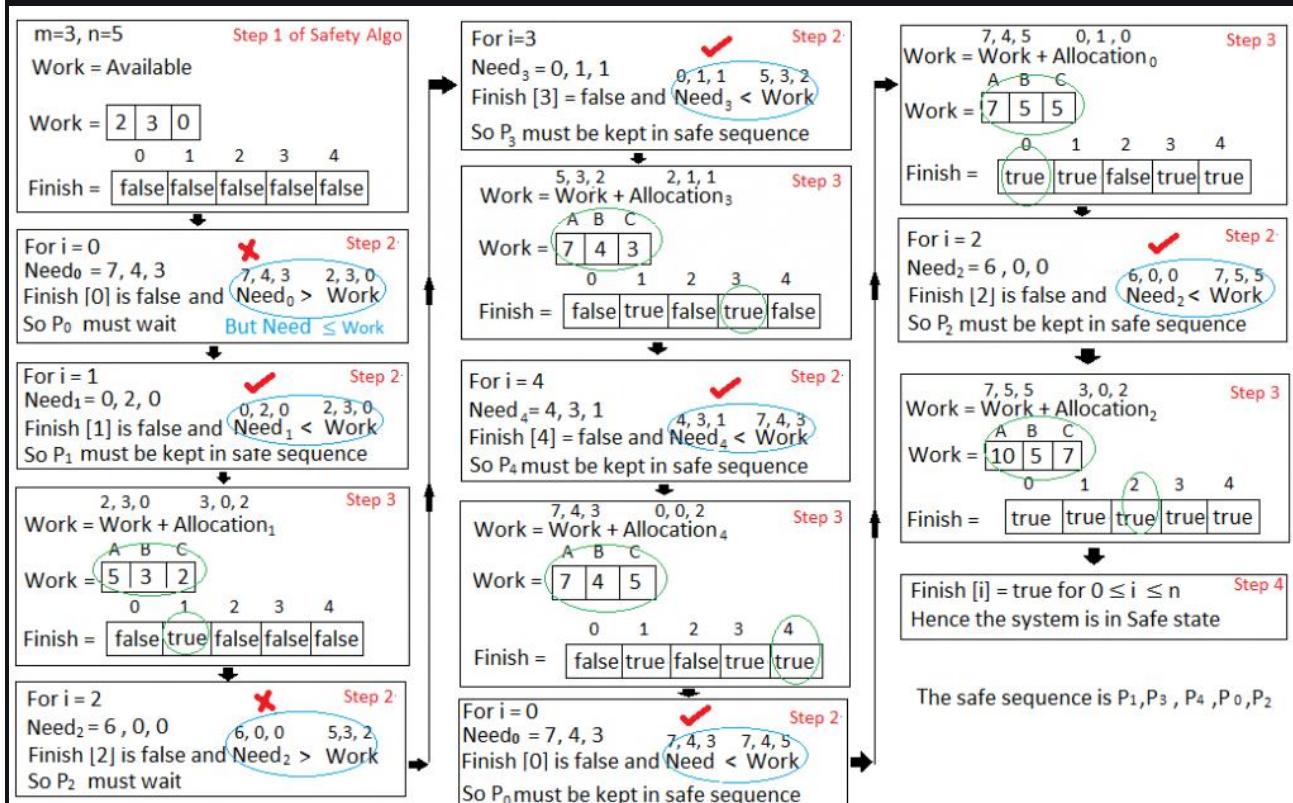
Available = Available – Request₁ Step 3

Allocation₁ = Allocation₁ + Request₁

Need₁ = Need₁ - Request₁

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	4	3	2 3 0	2 3 0	2 3 0
P ₁	3	0	2	0	2	0			
P ₂	3	0	2	6	0	0			
P ₃	2	1	1	0	1	1			
P ₄	0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process P₁.

Code for Banker's Algorithm

• C

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0      // Allocation Matrix
                       { 2, 0, 0 }, // P1
                       { 3, 0, 2 }, // P2
                       { 2, 1, 1 }, // P3
                       { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0      // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
```

```

    if (f[i] == 0) {

        int flag = 0;
        for (j = 0; j < m; j++) {
            if (need[i][j] > avail[j]){
                flag = 1;
                break;
            }
        }

        if (flag == 0) {
            ans[ind++] = i;
            for (y = 0; y < m; y++)
                avail[y] += alloc[i][y];
            f[i] = 1;
        }
    }
}

int flag = 1;

for(int i=0;i<n;i++)
{
    if(f[i]==0)
    {
        flag=0;
        printf("The following system is not safe");
        break;
    }
}

if(flag==1)
{
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
}
}

return (0);

// This code is contributed by Deep Baldha (CandyZack)

```

}

Output

Following is the SAFE Sequence

P1 → P3 → P4 → P0 → P2

As [Banker's algorithm](#) using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those table, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called

7.6 Resource Allocation Graph (RAG)

A resource allocation graphs shows which resource is held by which process and which process is waiting for a resource of a specific kind. It is amazing and straight – forward tool to outline how interacting processes can deadlock. Therefore, resource allocation graph describe what the condition of the system as far as process and resources are concern like what number of resources are allocated and what is the request of each process. Everything can be represented in terms of graph. One of the benefit of having a graph is, sometimes it is conceivable to see a deadlock straight forward by utilizing RAG and however you probably won't realize that by taking a glance at the table. Yet tables are better if the system contains bunches of process and resource and graph is better if the system contains less number of process and resource.

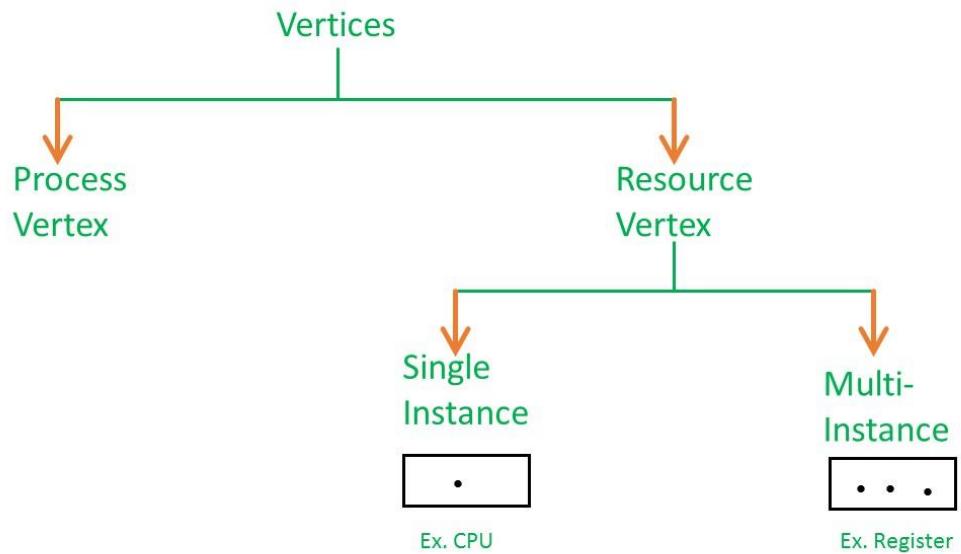
. So, resource allocation graph is explained to us what is the state of the system in terms of **processes and resources**. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges.

Types of Vertices in RAG

So RAG also contains vertices and edges. In RAG vertices are two types

1. **Process Vertex:** Every process will be represented as a process vertex. Generally, the process will be represented with a circle.
2. **Resource Vertex:** Every resource will be represented as a resource vertex. It is also two types:

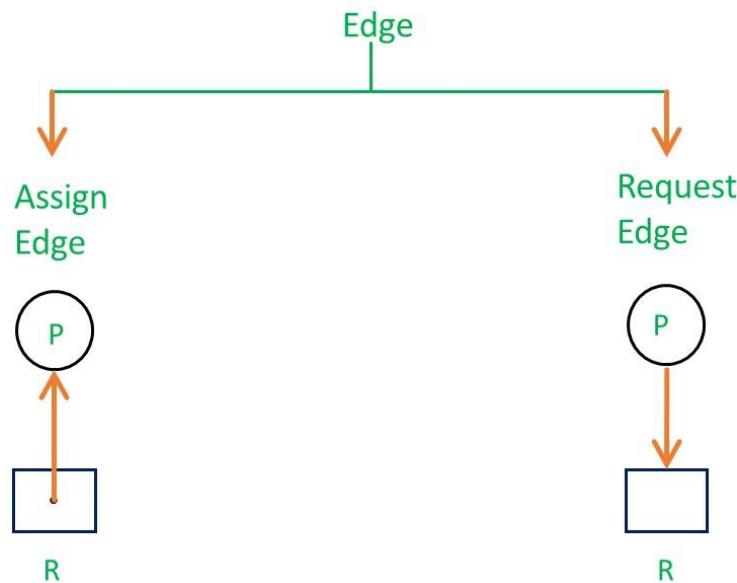
- **Single instance type resource:** It represents as a box, inside the box, there will be one dot. So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource:** It also represents as a box, inside the box, there will be many dots present.



How many Types of Edges are there in RAG?

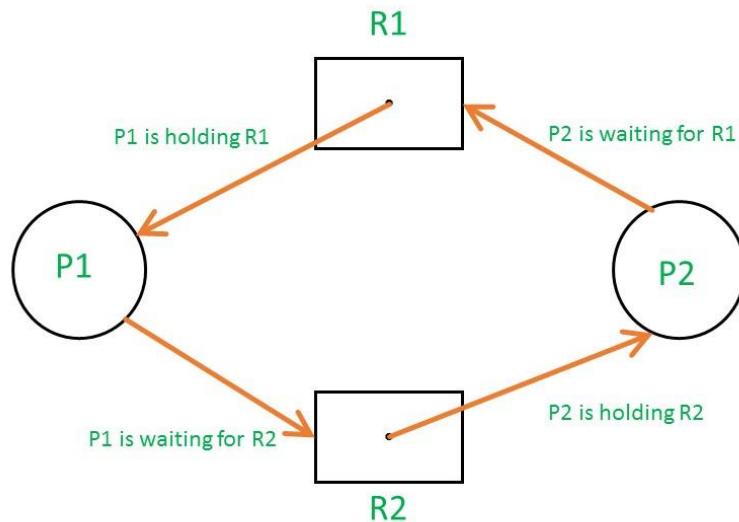
Now coming to the edges of RAG. There are two types of edges in RAG –

- **Assign Edge:** If you already assign a resource to a process then it is called Assign edge.
- **Request Edge:** It means in future the process might want some resource to complete the execution, that is called request edge.



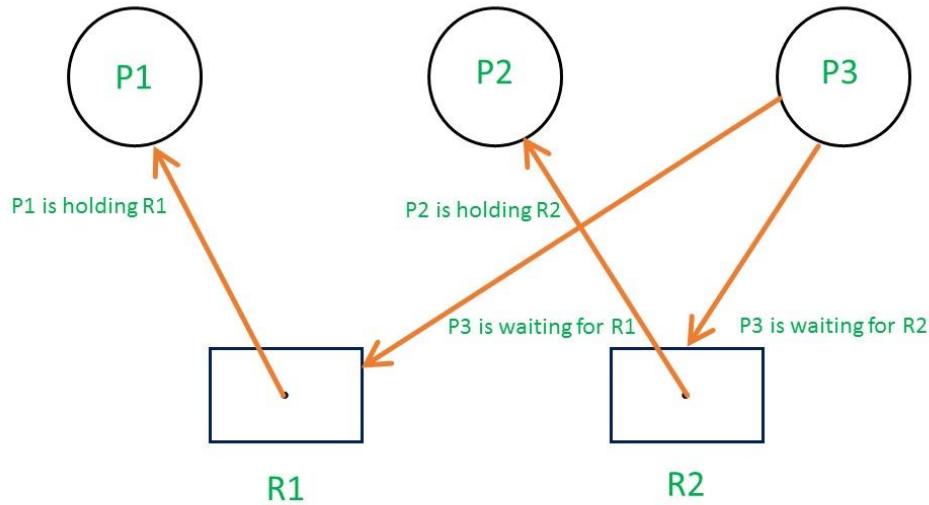
So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG)



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

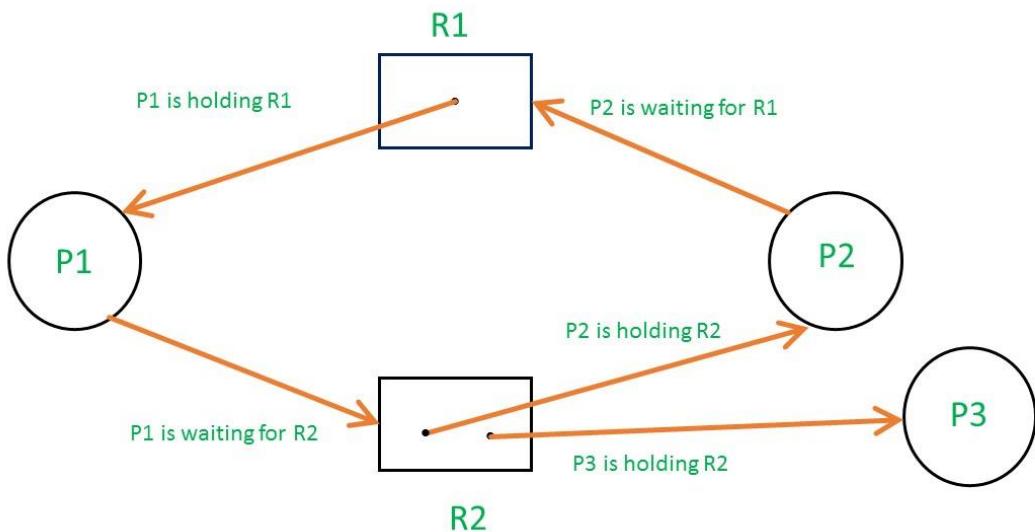
If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.



SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG)



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.
- Allocation matrix –**
- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.
- Request matrix –**
- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.
- So now available resource is = (0, 0).
- Checking deadlock (safe or not) –**

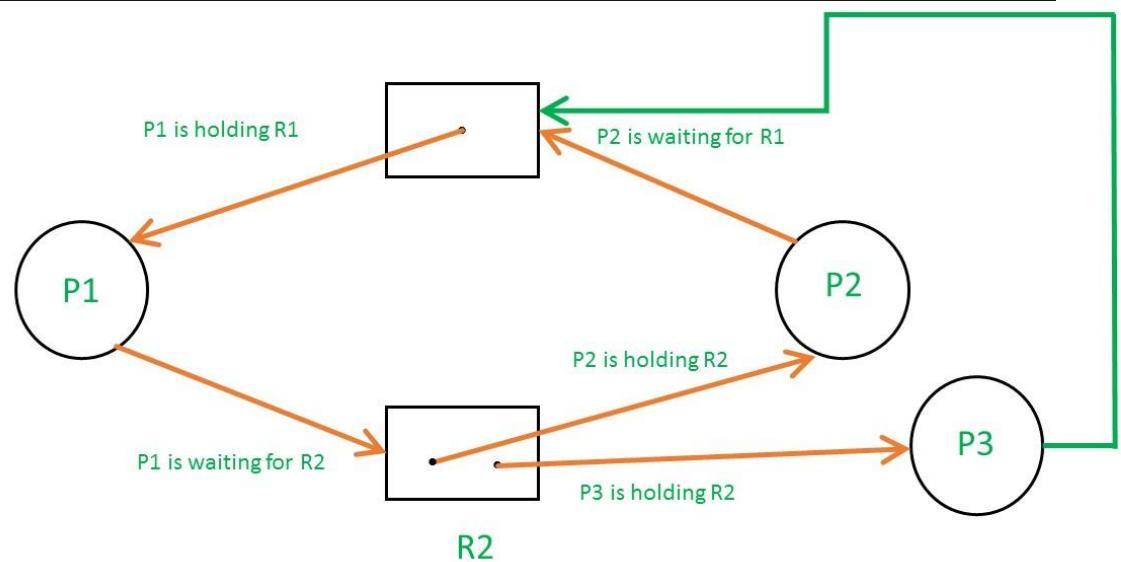
Available = 0 0 (As P3 does not require any extra resource to complete the execution and after completion P3 release its own resource)

New Available = 0 1 (As using new available resource we can satisfy the requirement of process P1)
P1 1 0 and P1 also releases its previous resource)

New Available = 1 1 (Now easily we can satisfy the requirement of process P2)
P2 0 1

New Available = 1 2

- So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore in multi-instance resource cycle is not sufficient condition for deadlock.



MULTI INSTANCES WITH DEADLOCK

- Above example is the same as the previous example except that, the process P3 is requesting for resource R1. So the table becomes as shown in below.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

- So, the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0). So you can't fulfill any one requirement. Therefore, it is in deadlock. Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.
- The Operating System allocates resources when a program need them. When the program terminates, the resources are de-allocated, and allocated to other programs that need them. Now the question is, what strategy does the operating system use to allocate these resources to user programs?
- There are two Resource allocation techniques:
 - **1. Resource partitioning approach** – In this approach, the operating system decides beforehand, that what resources should be allocated to which user program. It divides the resources in the system to many *resource partitions*, where each partition may include various resources – for example, 1 MB memory, disk blocks, and a printer. Then, it allocates one resource partition to each user program before the program's initiation. A resource table records the resource partition and its current allocation status (Allocated or Free).
 - **Advantages:**
 - Easy to Implement
 - Less Overhead
 - **Disadvantages:**
 - **Lacks flexibility** – if a resource partition contains more resources than what a particular process requires, the additional resources are wasted.

- If a program needs more resources than a single resource partition, it cannot execute (Though free resources are present in other partitions).

An Example Resource Table may look like this:-

Resource Table

Resource Name	Class	Allocation Status
Disk1	Disk	Allocated to P2
Printer2	Printer	Free
Disk 2	Disk	Free

The above Resource Table is of the time of Booting. Resource Table is the core data structure that is used in the resource allocation techniques. Resource Table contains resource partitions as entries rather than single resources.

2. Pool based approach – In this approach, there is a *common pool of resources*. The operating System checks the allocation status in the resource table whenever a program makes a request for a resource. If the resource is free, it allocates the resource to the program.

Advantages:

- Allocated resources are not wasted.
- Any resource requirement can be fulfilled if the resource is free (unlike Partitioning approach)

Disadvantages:

- Overhead of allocating and de-allocating the resources on every request and release.

Bankers algorithm:

```
// Java program to illustrate Banker's Algorithm
import java.util.*;
```

```
class GFG
{
    // Number of processes
    static int P = 5;

    // Number of resources
    static int R = 3;

    // Function to find the need of each process
    static void calculateNeed(int need[][], int maxm[][],
        int allot[][])
```

```

    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)
            // Need of instance = maxm instance -
            // allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
static boolean isSafe(int processes[], int avail[], int maxm[][],
                      int allot[][])
{
    int [][]need = new int[P][R];

    // Function to calculate need matrix
    calculateNeed(need, maxm, allot);

    // Mark all processes as infinish
    boolean []finish = new boolean[P];

    // To store safe sequence
    int []safeSeq = new int[P];

    // Make a copy of available resources
    int []work = new int[R];
    for (int i = 0; i < R ; i++)
        work[i] = avail[i];

    // While all processes are not finished
    // or system is not in safe state.
    int count = 0;
    while (count < P)
    {
        // Find a process which is not finish and
        // whose needs can be satisfied with current
        // work[] resources.
        boolean found = false;
        for (int p = 0; p < P; p++)
        {
            // First check if a process is finished,
            // if no, go for next condition
            if (finish[p] == false)
            {
                // Check if for all resources of
                // current P need is less
                // than work
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;
            }
            // If all needs of p were satisfied.
            if (j == R)
            {
                // Add the allocated resources of
                // current P to the available/work

```

```

        // resources i.e.free the resources
        for (int k = 0 ; k < R ; k++)
            work[k] += allot[p][k];

        // Add this process to safe sequence.
        safeSeq[count++] = p;

        // Mark this p as finished
        finish[p] = true;

        found = true;
    }
}

// If we could not find a next process in safe
// sequence.
if (found == false)
{
    System.out.print("System is not in safe state");
    return false;
}
}

// If system is in safe state then
// safe sequence will be as below
System.out.print("System is in safe state.\nSafe"
    +" sequence is: ");
for (int i = 0; i < P ; i++)
    System.out.print(safeSeq[i] + " ");

return true;
}

// Driver code
public static void main(String[] args)
{
    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources
    int avail[] = {3, 3, 2};

    // Maximum R that can be allocated
    // to processes
    int maxm[][] = {{7, 5, 3},
                    {3, 2, 2},
                    {9, 0, 2},
                    {2, 2, 2},
                    {4, 3, 3}};

    // Resources allocated to processes
    int allot[][] = {{0, 1, 0},
                    {2, 0, 0},
                    {3, 0, 2},
                    {2, 1, 1},
                    {0, 0, 2}};

    // Check system is in safe state or not

```

```
    isSafe(processes, avail, maxm, allot);
}
}
```

In the centralized approach of deadlock detection, **two techniques** are used namely: Completely centralized algorithm and Ho Ramamurthy algorithm (One phase and Two-phase).

- **Completely Centralized Algorithm –**

In a network of n sites, one site is chosen as a control site. This site is responsible for deadlock detection. It has control over all resources of the system. If a site requires a resource it requests the control site, the control site allocates and de-allocates resources, and maintains a wait-for graph. And at a regular interval of time, it checks the wait-for graph to detect a cycle. If the cycle exits then it will declare the system as deadlock otherwise the system will continue working. The major drawbacks of this technique are as follows:

1. A site has to send requests even for using its own resource.
2. There is a possibility of phantom deadlock.

- **HO Ramamurthy (Two-Phase Algorithm) –**

In this technique a resource status table is maintained by the central or control site, if a cycle is detected then the system is not declared deadlock at first, the cycle is checked again as the system is distributed some of the other resources is vacant or freed by sites at every instant of time. Now, after checking if a cycle is detected again then, the system is declared as deadlock. This technique reduces the possibility of phantom deadlock but on the other hand time consumption is more.

- **HO Ramamurthy (One Phase Algorithm) –**

In this technique a resource status table and a process table is maintained by the central or control site if the cycle is detected in both processes and resource tables then, the system is declared as deadlock. This technique reduces time consumption but space complexity increases.