Two Views of Operating System

1. User's View
2. System View

Operating System: User View

The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In these cases, the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.

Operating System: System View

Operating system can be viewed as a resource allocator also. A computer system consists of many resources like - hardware and software - that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls execution of programs etc.

Operating System Management Tasks

1. **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
2. **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
3. **Device management** which provides interface between connected devices.
4. **Storage management** which directs permanent data storage.
5. **Application** which allows standard communication between software and your computer.
6. **User interface** which allows you to communicate with your computer.

Functions of Operating System

1. It boots the computer
2. It performs basic computer tasks e.g. managing the various peripheral devices e.g. mouse, keyboard
3. It provides a user interface, e.g. command line, graphical user interface (GUI)
4. It handles system resources such as computer's memory and sharing of the central processing unit (CPU) time by various applications or peripheral devices.
5. It provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.
6. Error Handling is done by the operating system. It takes preventive measures whenever required to avoid errors.

Types of Operating Systems

Following are some of the most widely used types of Operating system.

1. Simple Batch System
2. Multiprogramming Batch System
3. Multiprocessor System
4. Desktop System
5. Distributed Operating System
6. Real-time Operating System
7. Handheld System

Simple Batch Systems

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

Advantages of Simple Batch Systems

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

Multiprogramming Batch Systems

- In this case, the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk (Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.

- In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.

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

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

Multiprocessor Systems

A Multiprocessor system consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

Advantages of Multiprocessor Systems

1. Enhanced performance
2. Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
3. If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.

Desktop Systems

Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither **multiuser** nor **multitasking**. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems are called **Desktop Systems** and include PCs running `Microsoft Windows` and the `Apple Macintosh`. Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.

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

Distributed Operating System

The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.

These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

Advantages Distributed Operating System

1. As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
2. Fast processing.
3. Less load on the Host Machine.

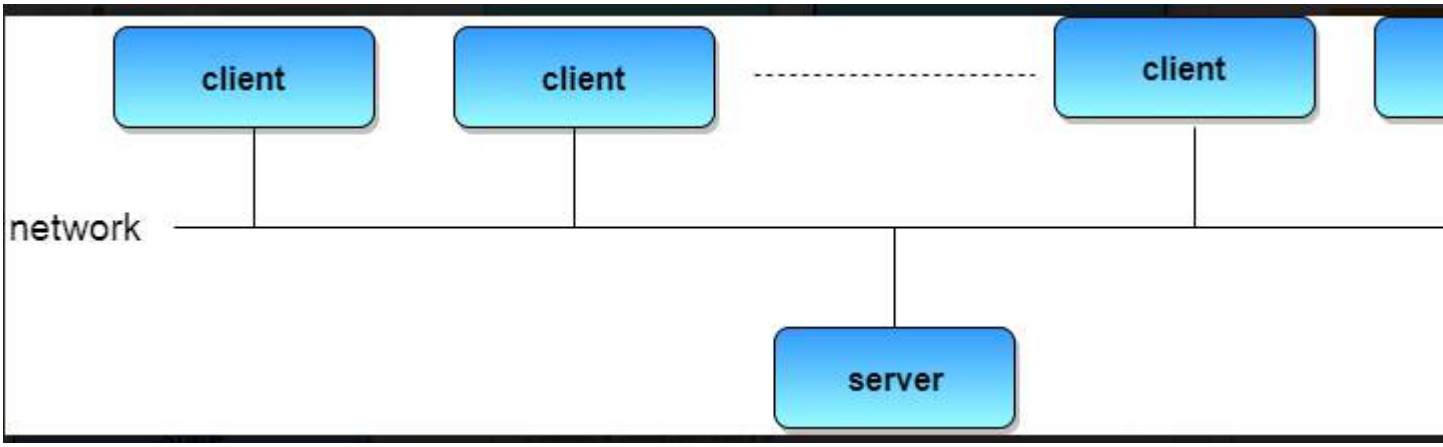
Types of Distributed Operating Systems

Following are the two types of distributed operating systems used:

1. Client-Server Systems
2. Peer-to-Peer Systems

Client-Server Systems

Centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client-server system is depicted in the figure below:



Server Systems can be broadly categorized as: **Compute Servers** and **File Servers**.

- **Compute Server systems**, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File Server systems**, provide a file-system interface where clients can create, update, read, and delete files.

Peer-to-Peer Systems

The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1990s for electronic mail and FTP, many PCs became connected to computer networks.

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

Real Time Operating System

It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.

The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems.**

While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurance of completing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

Handheld Systems

Handheld systems include **Personal Digital Assistants (PDAs)**, such as Palm-Pilots or **Cellular Telephones** with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

- Many handheld devices have between **512 KB** and **8 MB** of memory. As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used.
- Currently, many handheld devices do **not use virtual memory** techniques, thus forcing program developers to work within the confines of limited physical memory.
- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices may use wireless technology such as **Bluetooth**, allowing remote access to e-mail and web browsing. **Cellular telephones** with connectivity to the Internet fall into this category. Their use continues to expand as network connections become more available and other options such as `cameras` and `MP3 players`, expand their utility.

What is a Process?

A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working:

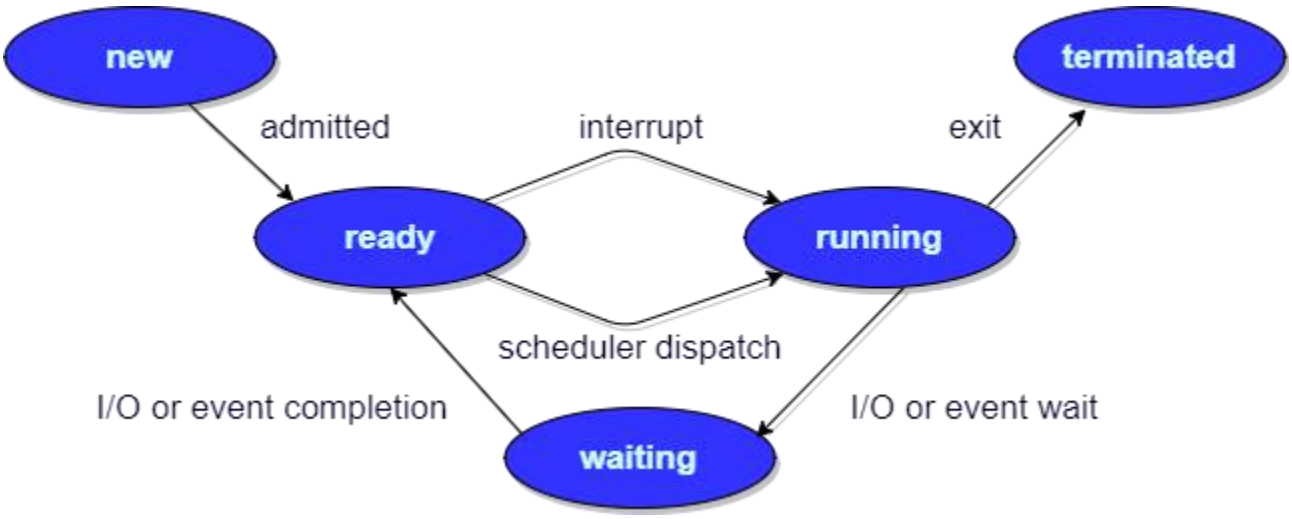
- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

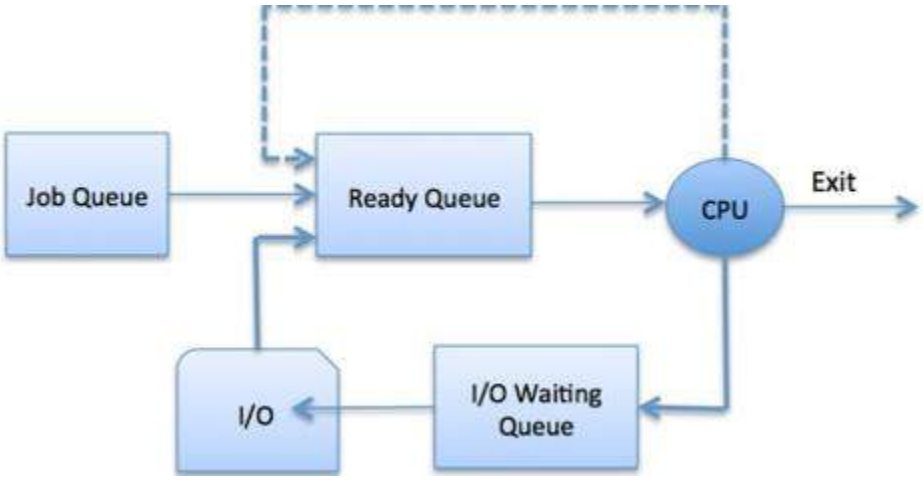
Different Process States

Processes in the operating system can be in any of the following states:

- NEW- The process is being created.
- READY- The process is waiting to be assigned to a processor.
- RUNNING- Instructions are being executed.

- WAITING- The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- TERMINATED- The process has finished execution.





Process Control Block

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:

- **Process State**: It can be running, waiting etc.
- **Process ID** and the **parent process ID**.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.

- **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
- **Memory Management information**: For example, page tables or segment tables.
- **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information**: Devices allocated, open file tables, etc.

What is Process Scheduling?

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

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

Job scheduling

---is the process where OS allocates system resources to many different tasks. The system handles prioritized **job** queues that are awaiting for CPU time and it should determine which **job** to be taken from the queue and the amount of time to be allocated for the **job**.

The objectives of a good scheduling policy include:

- Fairness.
- Efficiency.
- Minimum response time (important for interactive jobs).
- Minimum turnaround time (important for batch jobs).
- High throughput

Scheduling fall into one of the two general categories:

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

What are Scheduling Queues?

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

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

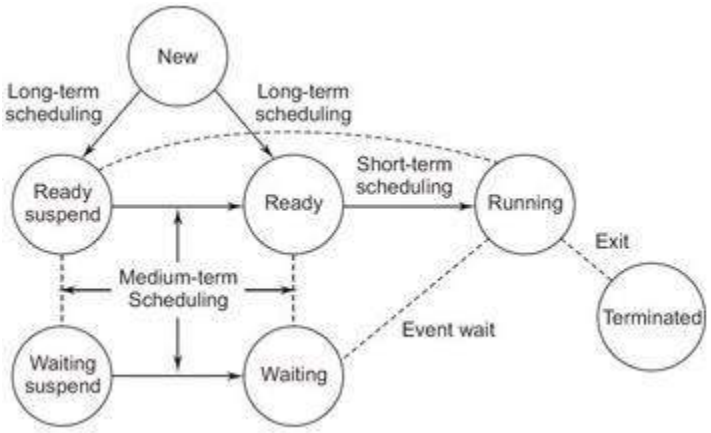
- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new sub-process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

The objective **of multiprogramming** is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently. In uniprocessor only one process is running. A process migrates between various scheduling queues throughout its lifetime. The process of selecting processes from among these queues is carried out by a **scheduler**. The aim of processor scheduling is to assign processes to be executed by the processor. Scheduling affects the performance of the system, because it determines which process will wait and which will progress.

Types of Scheduling:

Long-term Scheduling

Long term scheduling is performed when a new process is created. It is shown in the figure below. If the number of ready processes in the ready queue becomes very high, then there is an overhead on the operating system (i.e., processor) for maintaining long lists, context switching and dispatching increases. Therefore, allow only limited number of processes in to the ready queue. The "long-term scheduler" managers this. Long-term scheduler determines which programs are admitted into the system for processing. Once when admit a process or job, it becomes process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler scheduling manage queues to minimize queueing delay and to optimize performance.



The long-term scheduler limits the number of processes to allow for processing by taking the decision to add one or more new jobs, based on FCFS (First-Come, first-serve) basis or priority or execution time or **Input/output** requirements. Long-term scheduler executes relatively infrequently.

Medium-term Scheduling

Medium-term scheduling is a part of the swapping function. When part of the main memory gets freed, the operating system looks at the list of suspend ready processes, decides which one is to be swapped in (depending on priority, memory and other resources required, etc). This scheduler works in close conjunction with the long-term scheduler. It will perform the swapping-in function among the swapped-out processes. Medium-term scheduler executes somewhat more frequently.

Short-term Scheduling

Short-term scheduler is also called as **dispatcher**. Short-term scheduler is invoked whenever an event occurs, that may lead to the interruption of the current running process. For example clock interrupts, I/O interrupts, operating system calls, signals, etc. Short-term scheduler executes most frequently. It selects from among the processes that are ready to execute and allocates the CPU to one of them. It must select a new process for the CPU frequently. It must be very fast.

Scheduling Criteria

Scheduling criteria is also called as scheduling methodology. Different CPU scheduling algorithm have different properties .The criteria used for comparing these algorithms include the following:

- **CPU Utilization:**

  Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it range from 40 to 90%.

- **Throughput:**

  Throughput is the rate at which processes are completed per unit of time.

- **Turnaround time:**

  This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

- **Waiting time:**

  Waiting time is the sum of the time periods spent in waiting in the ready queue.

- **Response time:**

  Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

- **Fairness:**

  Each process should have a fair share of CPU.

Non-preemptive Scheduling:

In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/output or by requesting some operating system service.

Preemptive Scheduling:

In preemptive mode, currently running process may be interrupted and moved to the ready State by the operating system.
When a new process arrives or when an interrupt occurs, preemptive policies may incur greater overhead than non-preemptive version but preemptive version may provide better service.

It is desirable to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time and response time.

Scheduling Algorithms

Scheduling algorithms or scheduling policies are mainly used for short-term scheduling. The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior.
For these scheduling algorithms assume only a single processor is present. Scheduling algorithms decide which of the processes in the ready queue is to be allocated to the CPU is basis on the type of scheduling policy and whether that policy is either preemptive or non-preemptive.

For scheduling arrival time and service time are also will play a role.
List of scheduling algorithms are as follows:

- **First-come, first-served scheduling (FCFS) algorithm**
- **Shortest Job First Scheduling (SJF) algorithm**
- **Shortest Remaining time (SRT) algorithm**
- **Non-preemptive priority Scheduling algorithm**
- **Preemptive priority Scheduling algorithm**
- **Round-Robin Scheduling algorithm**

First-come First-served Scheduling (FCFS)

- ✓ First-come First-served Scheduling follow **first in, first out** method.
- ✓ As each process becomes ready, it joins the ready queue.
- ✓ When the current running process ceases to execute, the oldest process in the Ready queue is selected for running.

That is first entered process among the available processes in the ready queue.

Advantages

- Better for long processes
- Simple method (i.e., minimum overhead on processor)
- No starvation

Disadvantages

- FCFS may suffer from the ***convoy effect***. ***Convoy effect*** is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time. This essentially leads to poor utilization of resources and hence poor performance.

- Throughput is not emphasized.

The average waiting time for FCFS is often quite long, and it is non-preemptive.

Calculation of Average Waiting Time

For every scheduling algorithm, **Average waiting time** is a crucial parameter to judge it's performance.

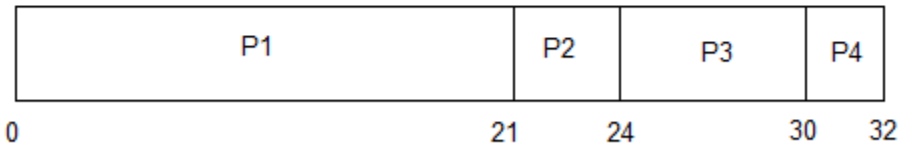
AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

Lower the Average Waiting Time, better the scheduling algorithm.

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with **Arrival Time** `0`, and given **Burst Time**, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

The average waiting time will be `18.75 ms`

For the above given proccesses, first **P1** will be provided with the CPU resources,

- Hence, waiting time for **P1** will be `0`
- **P1** requires `21 ms` for completion, hence waiting time for **P2** will be `21 ms`
- Similarly, waiting time for process **P3** will be execution time of **P1** + execution time for **P2**, which will be `(21 + 3) ms` = `24 ms`.
- For process **P4** it will be the sum of execution times of **P1**, **P2** and **P3**.

Shortest Job First Scheduling (SJF)

- ✓ This algorithm associates with each process the length of the next CPU burst. Shortest-job-first scheduling is also known as shortest process next (SPN).

- ✓ The process with the shortest expected processing time is selected for execution, among the available processes in the ready queue.
- ✓ Thus, a short process will jump to the head of the queue over long jobs.
- ✓ If the next CPU bursts of two processes are the same then FCFS scheduling is used to break the tie. It gives the minimum average time for a given set of processes.
- ✓ SJF can be preemptive or non-preemptive.
- ✓ A preemptive SJF algorithm will preempt the currently executing process if the next CPU burst of newly arrived process may be shorter than what is left to the currently executing process. Preemptive SJF Scheduling is sometimes called Shortest Remaining Time First algorithm.

- ✓ A Non-preemptive SJF algorithm will allow the currently running process to finish.

Advantages

- It gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
- Throughput is high.

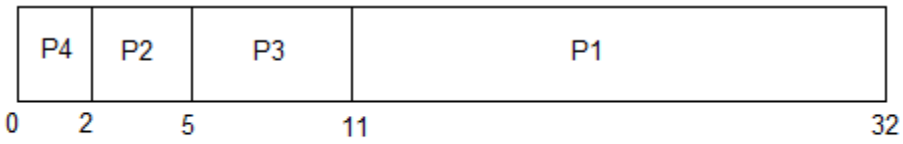
Disadvantages

- Elapsed time (i.e., execution-completed-time) must be recorded, it results an additional overhead on the processor.
- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Starvation may be possible for the longer processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT CHART will look like this:



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

With FCFS the average waiting time was **18.75** ms
And with the SJF, the average waiting time is **4.5** ms

Priority Scheduling

The SJF is a special case of general priority scheduling algorithm.
A Priority (an integer) is associated with each process. The CPU is allocated to the process with the highest priority. Generally smallest integer is considered as the highest priority. Equal priority processes are scheduled in First Come First serve order. It can be preemptive or Non-preemptive.

Non-preemptive Priority Scheduling

In this type of scheduling the CPU is allocated to the process with the highest priority after completing the present running process.

Problem with Non Pre-emptive SJF

If the **arrival times** for processes are different, which means all the processes are not available in the ready queue at time `0`, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non-Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

Advantage

- Good response for the highest priority processes.

Disadvantage

- Starvation may be possible for the lowest priority processes.

Preemptive Priority Scheduling

In this type of scheduling the CPU is allocated to the process with the highest priority immediately upon the arrival of the highest priority process. If the equal priority process is in running state, after the completion of the present running process CPU is allocated to this even though one more equal priority process is to arrive.

Advantage

- Very good response for the highest priority process over non-preemptive version of it.

Disadvantage

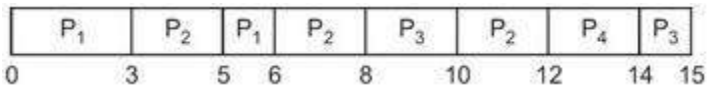
- Starvation may be possible for the lowest priority processes.

Round-Robin Scheduling

This type of scheduling algorithm is basically designed for time sharing system. It is similar to FCFS with preemption added. Round-Robin Scheduling is also known as time-slicing scheduling and it is a preemptive version based on a clock. That is a clock interrupt is generated at periodic intervals usually 10-100ms. When the interrupt occurs, the currently running process is placed in the ready queue and the next ready job is selected on a First-come, First-serve basis. This process is known as time-slicing, because each process is given a slice of time before being preempted. One of the following happens:

- The process may have a CPU burst of less than the time quantum or
- CPU burst of currently executing process be longer than the time quantum. In this case the context switch occurs and the process is put at the tail of the ready queue.

In **Round-robin** scheduling, the principal design issue is the length of the time quantum or time-slice to be used. If the quantum is very short, then short processes will move quickly.



Advantages

- Round-robin is effective in a general-purpose, times-sharing system or transaction-processing system.
- Fair treatment for all the processes.
- Overhead on processor is low.
- Good response time for short processes.

Disadvantages

- Care must be taken in choosing quantum value.
- Processing **overhead** is there in handling **clock** interrupt.
- Throughput is low if time quantum is too small.
- The performance of RR depends on time slice. If it is large then it is the same as FCFS. If q is small then overhead is too high.

Exercise

1. **Given the following two Gantt charts, calculate average waiting time in terms of FCFS and SJF. Draw the two diagrams for both.**

PROCESS	BURST TIME
P1	8
P2	4
P3	4
P4	4

2.

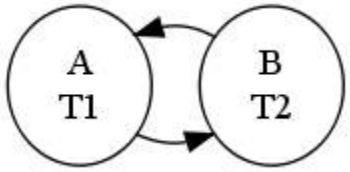
PROCESS	BURST TIME
P1	2
P2	1
P3	4
P4	3

Deadlock in Operating System.

A set of two or more processes are deadlocked if they are blocked (i.e., in the waiting state) each holding a resource and waiting to acquire a resource held by another process in the set.

or

A process is deadlocked if it is waiting for an event which is never going to happen.



Example:

- a system has two tape drives
- two processes are deadlocked if each holds one tape drive and has requested the other

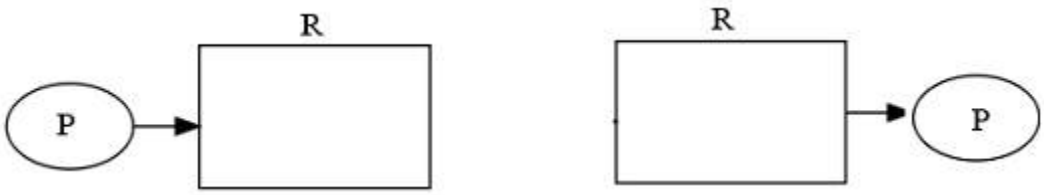
Conditions Necessary for Deadlock

All of the following four necessary conditions must hold simultaneously for deadlock to occur:

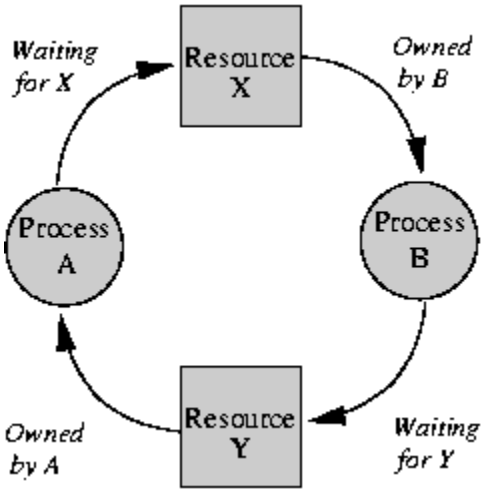
Necessary Conditions

- These are four conditions that are necessary to achieve a deadlock:
    1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
    2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
    3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
    4. **circular wait:** a cycle of process requests exists (i.e., P_0 is waiting for a resource hold by P_1 who is waiting for a resource held by P_j ... who is waiting for a resource held by $P_{(n-1)}$ which is waiting for a resource held by P_n which is waiting for a resource held by P_0).

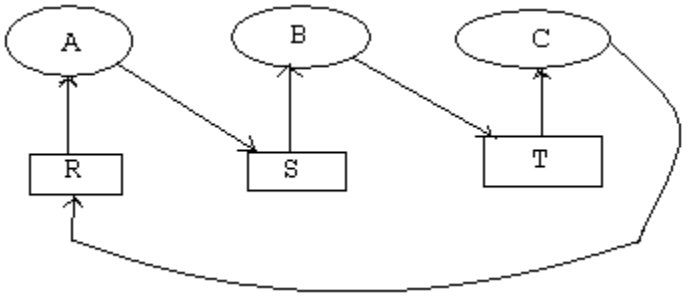
- Circular wait implies the hold and wait condition. Therefore, these conditions are not completely independent.



- Multiple independent requests: processes do not ask for resources all at once (**Hold and Wait**).
- There is circularity in the graph of who has what and who wants what. Usually a graph is drawn showing processes as circles, resources as squares, arrows from process to resource waited for, from resource to owning process, as shown in the graph below.

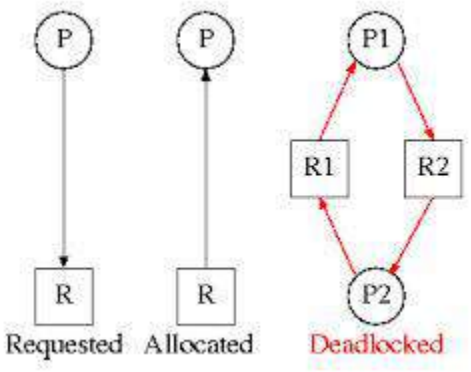


Another situation may look something like this diagram below, and many others:



In the diagram above, A, B and C represent processes and R, S and T are resources.

Deadlock and Starvation: Key Difference: Deadlock refers to the situation when processes are stuck in circular waiting for the resources. On the other hand, starvation occurs when a process waits for a resource indefinitely. Deadlock implies starvation but starvation does not imply deadlock.



In computer system memory printers, CPUs, tape drives, etc. can be considered as resources which need to be allocated to various processes due to their requirement. Generally, first a request is made by the process to use a resource, and after completion of its job, the process releases the resource to be used by some other process. A situation of deadlock arises when all the blocked processes of one set each occupies a resource and wait for the resource which is occupied by some other process in the set.

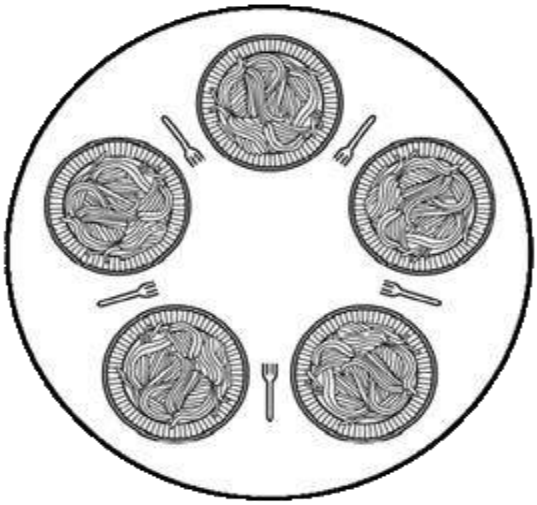
A simple example of it is a system with two tape drives, and two process each occupies one tape drive and waiting for the other as the requirement to proceed further.

Comparison between Deadlock and Starvation:

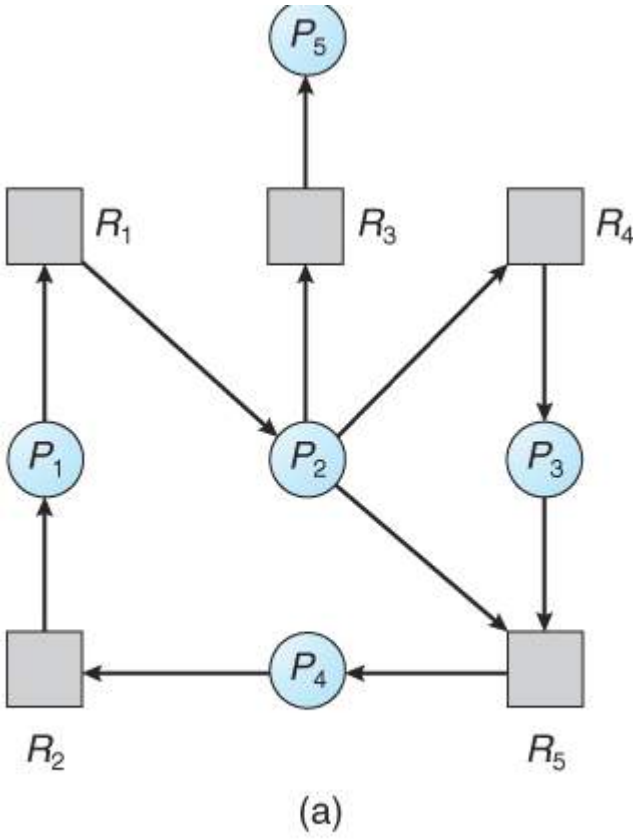
	Deadlock	**Starvation**
Definition	Deadlock occurs when none of the processes in the set is able to move ahead due to occupancy of the required resources by some other process	Starvation occurs when a process waits for an indefinite period of time to get the resource it requires.
Other name	Circular waiting	Lived lock
Arising conditions	These four conditions arising simultaneously – mutual exclusion, hold and wait, no-preemption and circular wit	Uncontrolled management of resources Process priorities being strictly enforced Use of random selection

		Scarcity of resources
Avoidance/ prevention Techniques	• Infinite resources • Waiting is not allowed • Sharing is not allowed • Preempt the resources • All Requests made at the starting	• Independent manager for each resources • No strict enforcement of the priorities • Avoidance of random selection • Providing more resources
Progress	No process can make progress	Apart from the victim process other processes can progress or proceed
Ending	Requires external intervention	May or may not require external intervention

The **Dining Philosophers Problem** (done by Edsger Dijkstra and Tony Hoare, 1965)

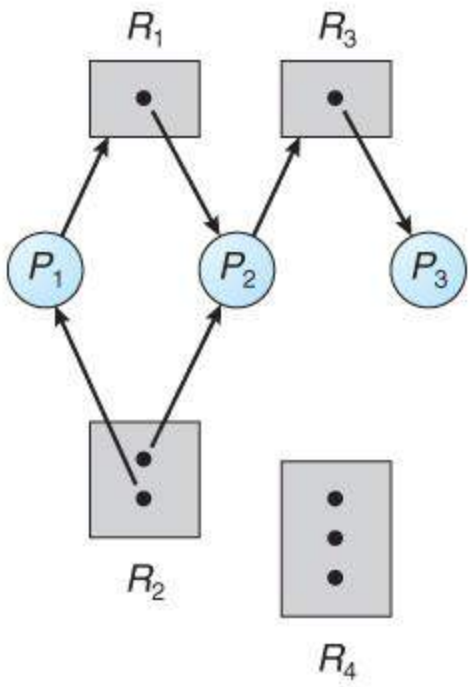


- Five philosophers sit at a circular table doing one of two things: *eating* or *thinking*.
- While eating, they are not thinking, and while thinking, they are not eating.
- A large bowl of spaghetti is in the center and a fork is placed in between each pair of adjacent philosophers.
- This spaghetti is difficult to serve and eat with a single fork, so each philosopher **must acquire two forks to eat**.
- Each philosopher **can only use the forks on their immediate left and right**.
- Each philosopher can acquire forks only **one at a time**.
- Philosophers never speak to each other.



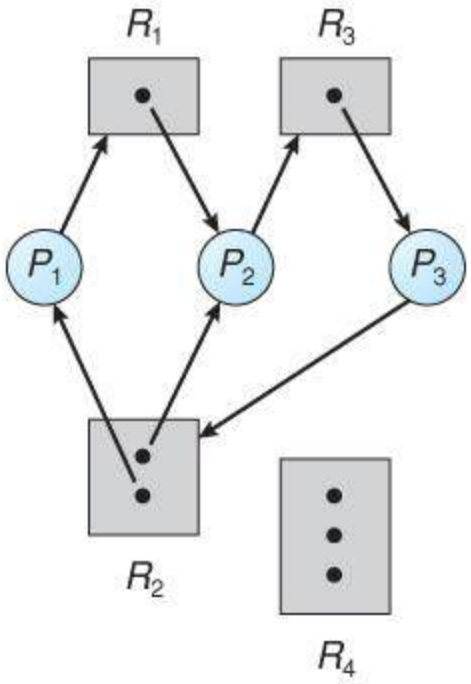
(a)

For example:



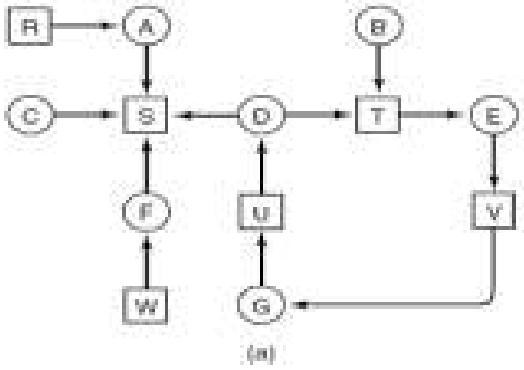
If a resource-allocation graph contains no cycles, then the system is not deadlocked.

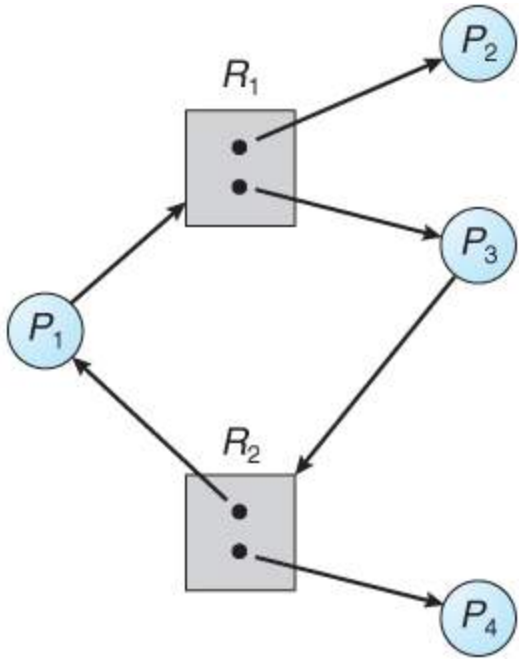
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.



Resource allocation graph with a deadlock

If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one.





Resource allocation graph with a cycle but no deadlock

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
    1. Deadlock **prevention** or **avoidance** - Do not allow the system to get into a deadlocked state.
    2. Deadlock **detection** and **recovery** - Abort a process or preempt some resources when deadlocks are detected.
    3. **Ignore** the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete

resource request and release plan for each process, depending on the particular algorithm.)

- Deadlock **detection** is fairly straightforward, but deadlock **recovery** requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown cannot be easily differentiated from a general system slowdown when a real-time process has heavy computing needs.

Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

- Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
  - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
  - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
  - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri
- One big challenge in this scheme is determining the relative ordering of the different resources

Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the four mentioned conditions.

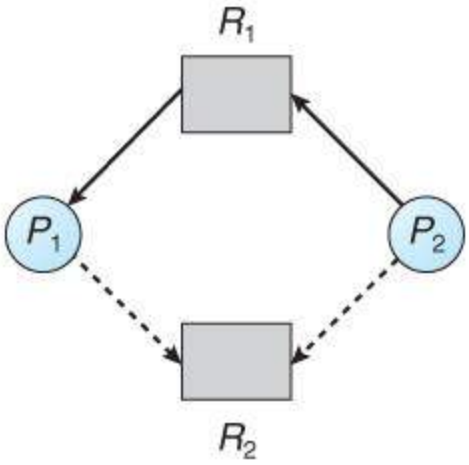
- This requires more information about each process, AND tends to lead to low device utilization. (i.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

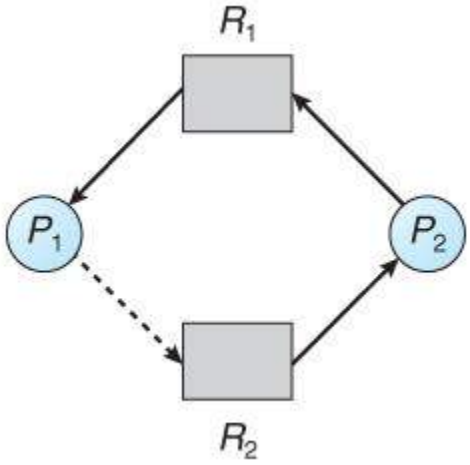
- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes {P0, P1, P2, ..., PN} such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (i.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

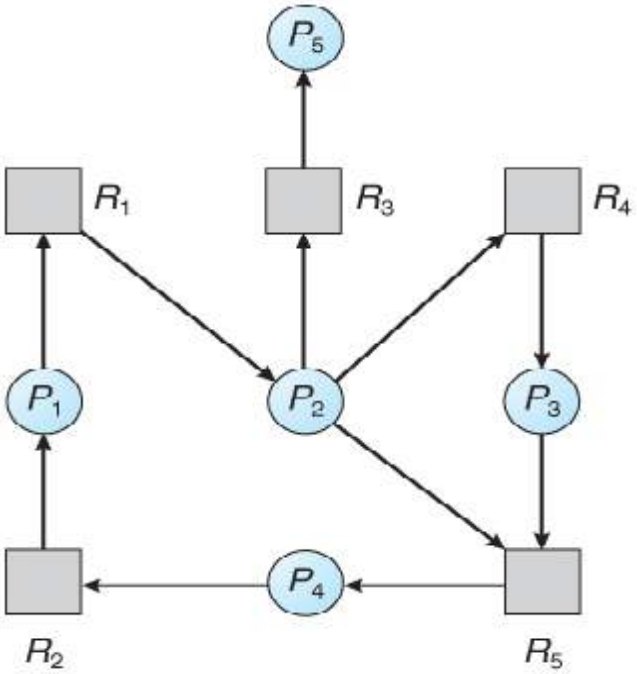
- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- Consider for example what happens when process P2 requests resource R2?

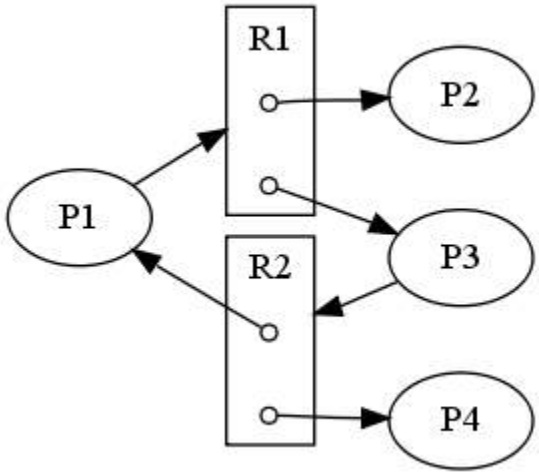


The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.



Unsafe state in a resource allocation graph





Recovery from Deadlock

- There are three basic approaches to recovery from deadlock:
    1. Inform the system operator, and allow him/her to take manual intervention.
    2. Terminate one or more processes involved in the deadlock

3. Preempt resources.

Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
    - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
    - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
    1. Process priorities.
    2. How long the process has been running, and how close it is to finishing.
    3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    4. How many more resources does the process need to complete.
    5. How many processes will need to be terminated
    6. Whether the process is interactive or batch.

Resource Preemption

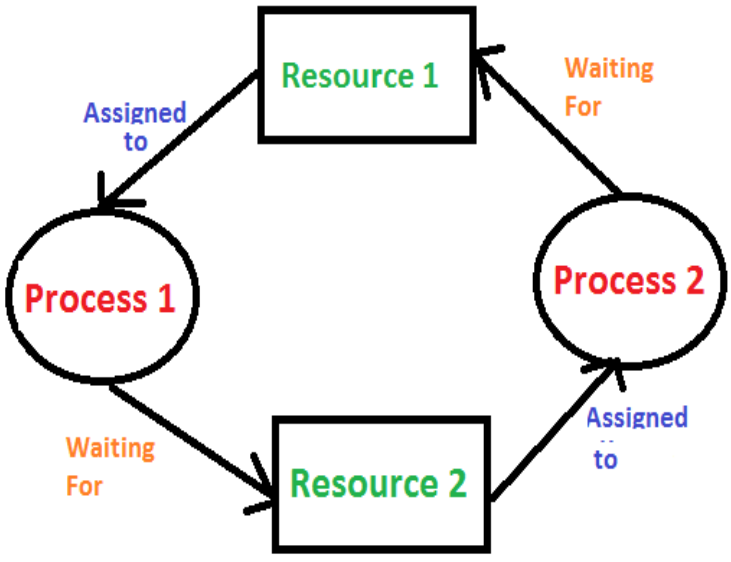
- When preempting resources to relieve deadlock, there are three important issues to be addressed:
    1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
    2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)

3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

Deadlocks

In computer science, **deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain.

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 same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes 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.



A process in operating systems uses different resources in the following ways:
1) Requests a resource
2) Use the resource
3) Releases the resource

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)
Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for other resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

How to avoid Deadlocks

Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock.

1. **Mutual Exclusion**

   Resources shared such as read-only files do not lead to deadlocks but resources, such as printers and tape drives, requires exclusive access by a single process.

2. **Hold and Wait**

   In this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

3. **No Preemption**

   Preemption of process resource allocations can avoid the condition of deadlocks, where ever possible.

4. **Circular Wait**

   Circular wait can be avoided if we number all resources, and require that processes request resources only in strictly increasing (or decreasing) order.

Handling Deadlock

The above points focus on preventing deadlocks. But what to do once a deadlock has occurred. Following three strategies can be used to remove deadlock after its occurrence.

1. **Preemption**

   We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

2. **Rollback**

   In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

3. **Kill one or more processes**

   This is the simplest way, but it works.

What is a Livelock?

There is a variant of deadlock called livelock. This is a situation in which two or more processes continuously change their state in response to changes in the other processes without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.

A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

Methods for handling deadlock:

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.
Avoidance is kind of futuristic in nature. By using strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

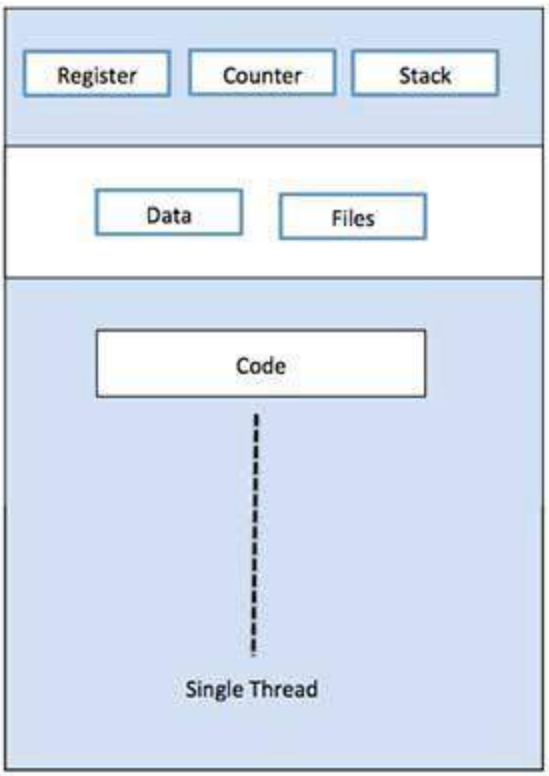
What is Thread?

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history. A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the **operating system**.

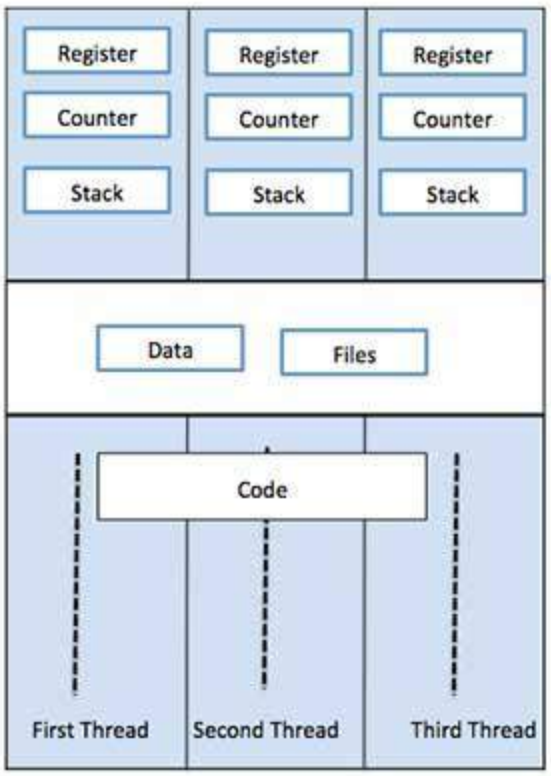
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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

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



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Threads

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

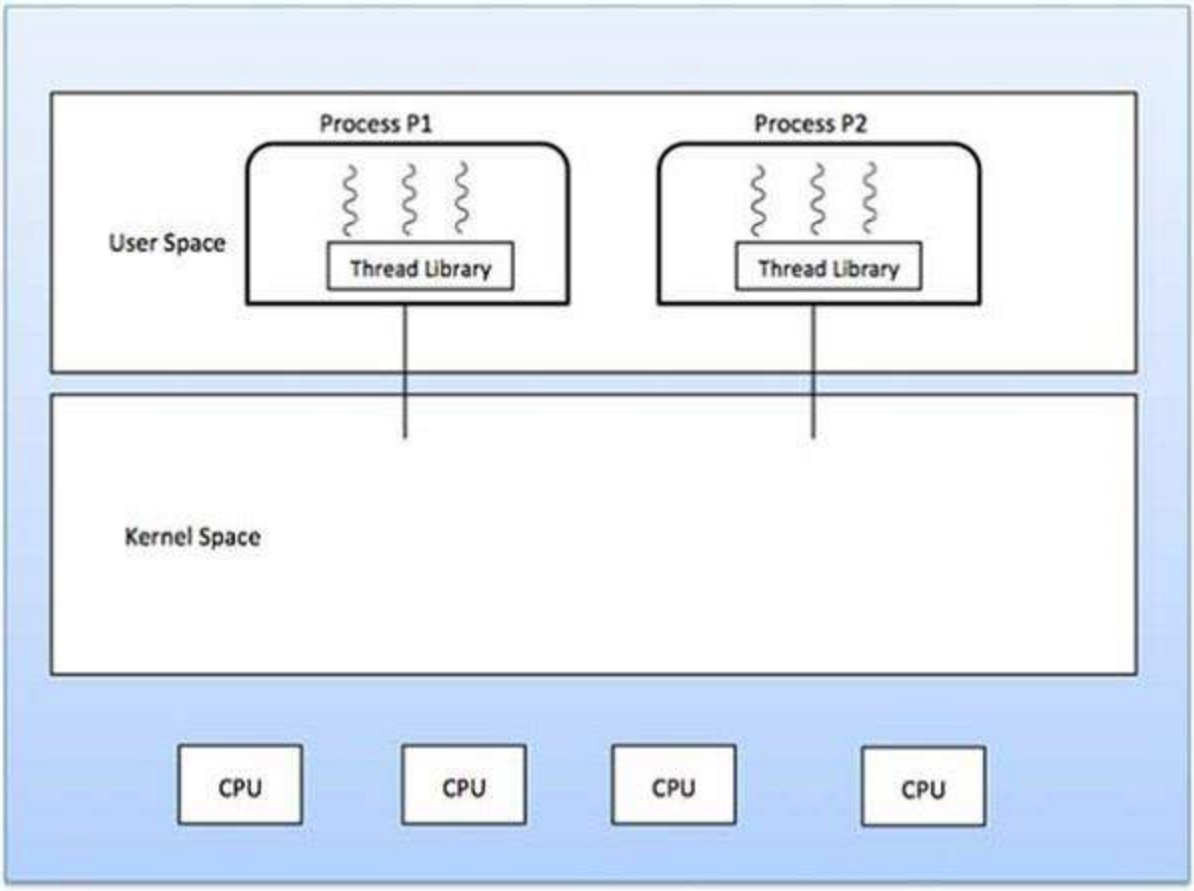
Types of Threads

Threads are implemented in following two ways −

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

User Level Threads

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



Advantages

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

Disadvantages

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

Kernel Level Threads

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

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

Advantages

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

Disadvantages

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

Multithreading Models

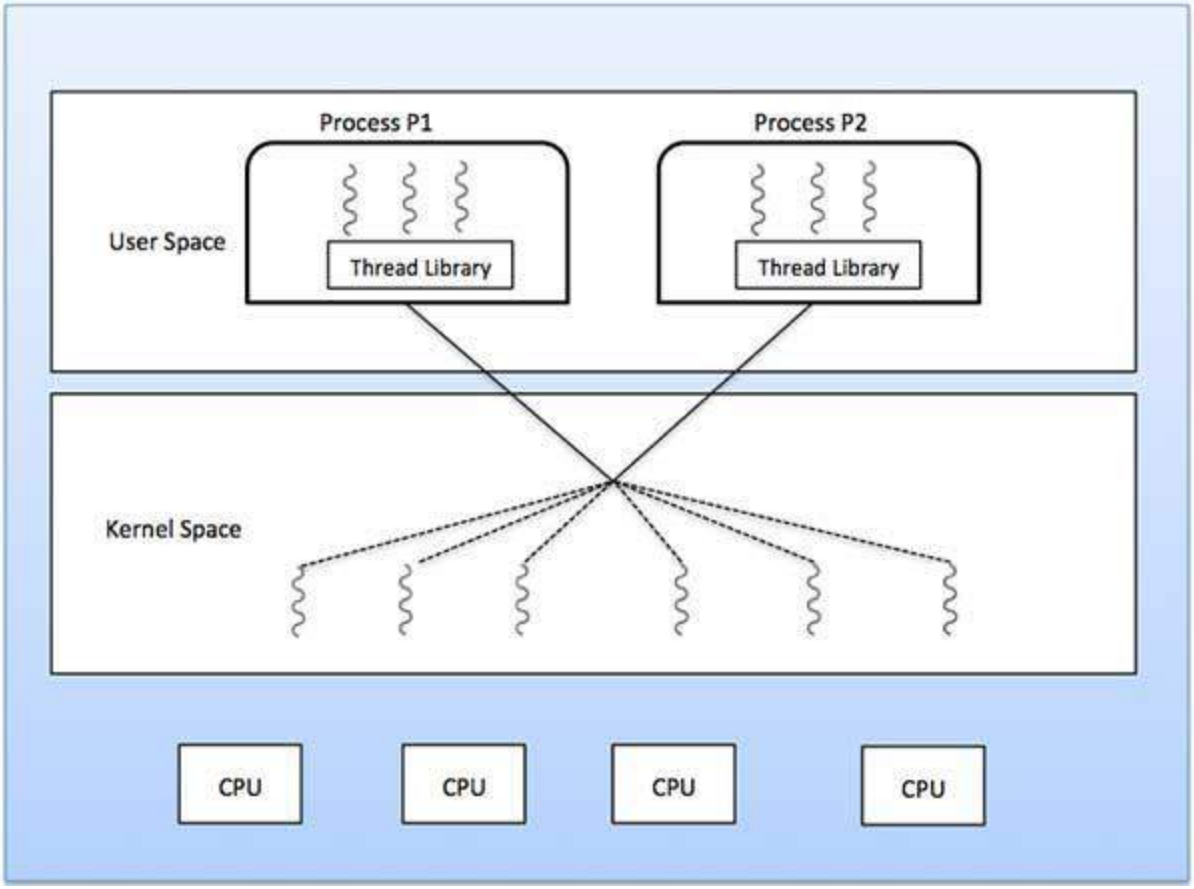
Some operating systems provide a combined user level thread and Kernel level thread facility. **Solaris** is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block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

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

Many to Many Model

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

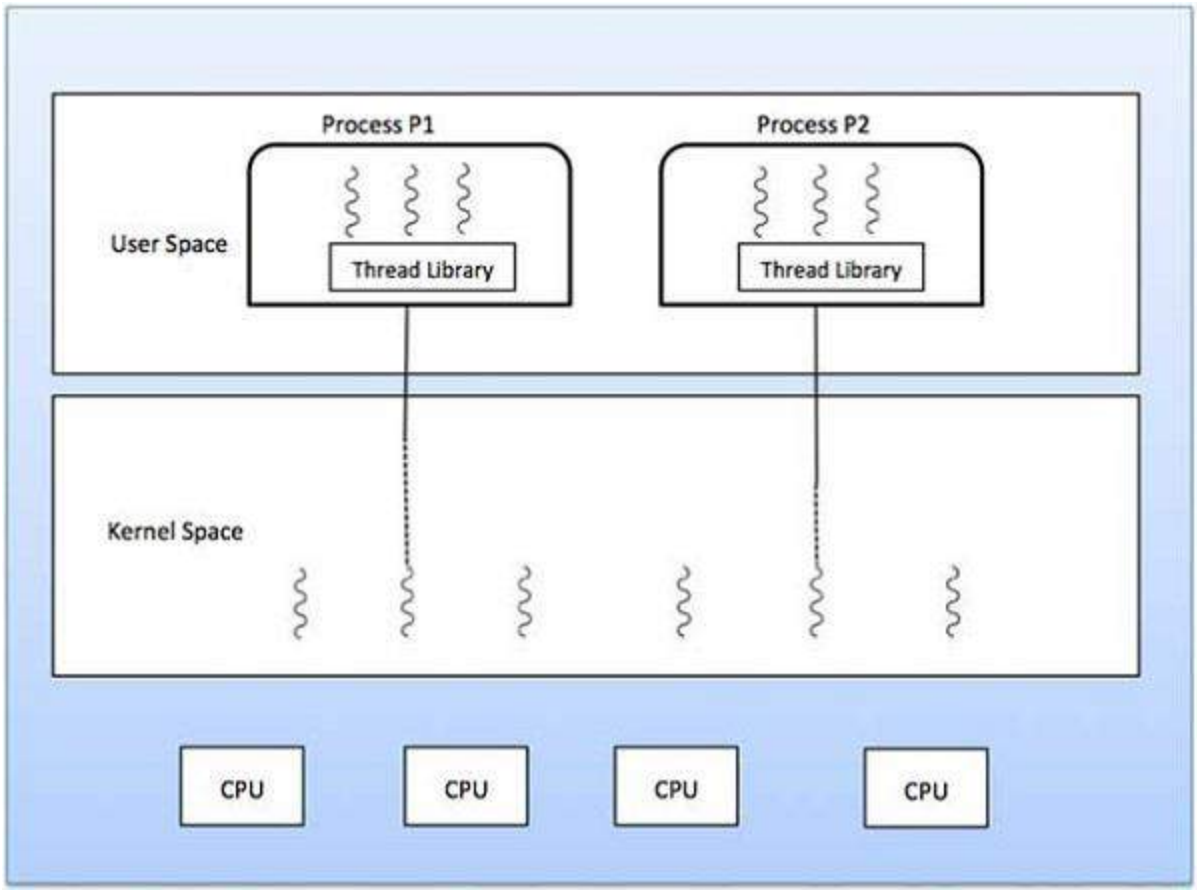
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model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



Many to One Model

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

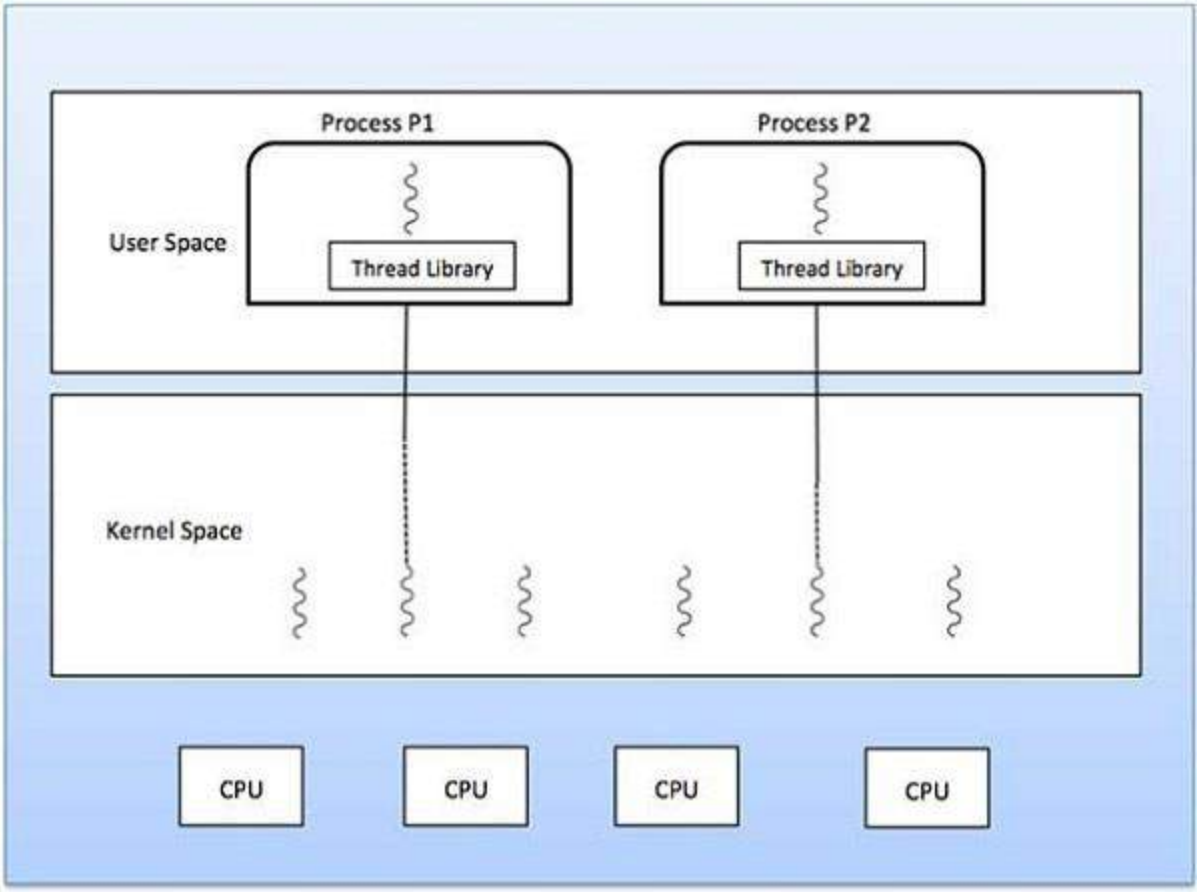
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

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



Difference between User-Level & Kernel-Level Thread

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

What is Memory?

Memory is very much like our brain as it is used to store data and instructions. Computer memory is the storage space where data is to be processed, and instructions needed for

processing are stored. The memory is divided into a large number of smaller portions called the cell. Every cell/ location has a unique address and a size.

Two types of memories are:

- **Primary Memory**
- **Secondary Memory**

What is Primary Memory?

Primary memory is the main memory of the computer system. Accessing data from primary memory is faster because it is an internal memory of the computer. The primary memory is most volatile which means data in primary memory does not exist if it is not saved when a power failure occurs.

The primary memory is a semiconductor memory. It is costlier compared with secondary memory. The capacity of primary memory is very much limited and is always smaller compares to secondary memory.

Two types of Primary Memory are:

- **RAM**
- **ROM**

KEY DIFFERENCE

- Primary memory is also called internal memory whereas Secondary memory is also known as a Backup memory or Auxiliary memory.
- Primary memory can be accessed by the data bus whereas Secondary memory is accessed by I/O channels.
- Primary memory data is directly accessed by the processing unit whereas Secondary memory data cannot be accessed directly by the processor.
- Primary memory is costlier than secondary memory whereas Secondary memory is cheaper compared to primary memory.
- Primary memory is both volatile & nonvolatile whereas Secondary memory is always a non-volatile memory.

RAM (Random Access Memory)

Random access memory which is also known as RAM is generally known as a main memory of the computer system. It is called temporary memory or cache memory. The information stored in this type of memory is lost when the power supply to the PC or laptop is switched off.

ROM (Read Only Memory)

It stands for Read Only Memory. ROM is a permanent type of memory. Its content is not lost when the power supply is switched off. The computer manufacturer decides the information of ROM, and it is permanently stored at the time of manufacturing which cannot be overwritten by the user.

What is Secondary Memory?

All secondary storage devices which are capable of storing high volume data is referred to secondary memory. It's slower than primary memory. However, it can save a substantial amount of data, in the range of gigabytes to terabytes. This memory is also called backup storage or mass storage media.

Types of Secondary memory

Mass storage devices:

The magnetic disk provides cheap storage and is used for both small and large computer systems.

Two types of magnetic disks are:

- Floppy disks
- Hard disks

Flash/SSD

Solid State Drive provides a persistent flash memory. A simple USB flash drive is an **example** of solid-state drive technology. It's very fast compared to Hard Drives.

Why solid state drives (SSDs) are better

Solid state drives (SSD), as the name suggests, don’t have moving parts or spinning disks.

They use interconnected pools of flash memory that are managed by an SSD controller to deliver speeds far beyond what an HDD can offer.

SSDs can reduce boot time from around 35 seconds to about 10 seconds.

Frequently found in Mobile phones, it’s rapidly being adopted in PC/Laptop/Mac.

How does an SSD work?

A simple USB flash drive (or thumb drive) is an example of solid-state drive technology. An SSD is a larger, more complex device that aggregates pools of NAND flash storage, the type of storage also found in MP3 players and digital cameras. Flash memory is an electronic non-volatile computer memory storage medium that can be electrically erased and reprogrammed. The two main types of flash memory, NOR flash and NAND flash, are named after the NOR and NAND logic gates. Unlike RAM, which doesn't retain data when the machine shuts off, SSD flash memory is non-volatile, which means data is retained whether the device is powered on or not.

With SSDs, every block of data is accessible at the same speed as every other block, no matter the location. This makes SSDs inherently faster than hard drives, where platters are spinning and drive heads are moving to the right location.

Intel SSD DC P6400 Series

Intel's SSD DC P6400 Series is a 3D NAND solid state drive designed for data centers

SSD vs. HDD (hard disk drives) in the enterprise

SSDs have a number of advantages over HDDs that can help offset the difference in sticker price.

SSDs are quiet.

They don't vibrate, which improves reliability.

If dropped, a hard drive might get damaged; not so with an SSD.

They use less power and generate less heat, which can add up to big savings in a large data center scenario.

They are also smaller and more powerful than HDDs, so data centers can pack more storage into less space.

There’s the speed advantage.

But despite their performance advantages, SSDs only have a 10% market share compared to HDDs for a couple of reasons. First and foremost, they’re expensive.

HDDs today average around 3-4 cents per GB, compared to 25-30 cents for SSDs.

SSDs get slower as they fill up. And eventually, the flash cells reach a state where they can no longer complete write operations at all.

Optical drives:

- This secondary storage device is from where data is read and written with the help of lasers. Optical disks can hold data up to 185TB.
- **Examples** of this **media** are compact **disk** read-only memory (**CD**-ROM), digital versatile **disk** read-only memory (DVD-ROM), digital versatile **disk** random access memory (DVD-RAM), write-once read-many (WORM) cartridges, erasable **optical** cartridges, and Removable Mass Storage (RMS) **media** which are removable **disk** (RDX)

USB drives:

A USB flash drive is a data storage device that includes flash memory with an integrated USB interface.

It is typically removable, rewritable and much smaller than an optical disc.

It is one of the most popular types of secondary storage device available in the market.

USB drives are removable, rewritable and are physically very small.

The capacity of USB drives is also increasing significantly as today 1TB pen drive is also available in the market.

Magnetic tape:

It is a serial access storage device which allows us to store a very high volume of data. Usually used for backups.

Characteristic of Primary Memory

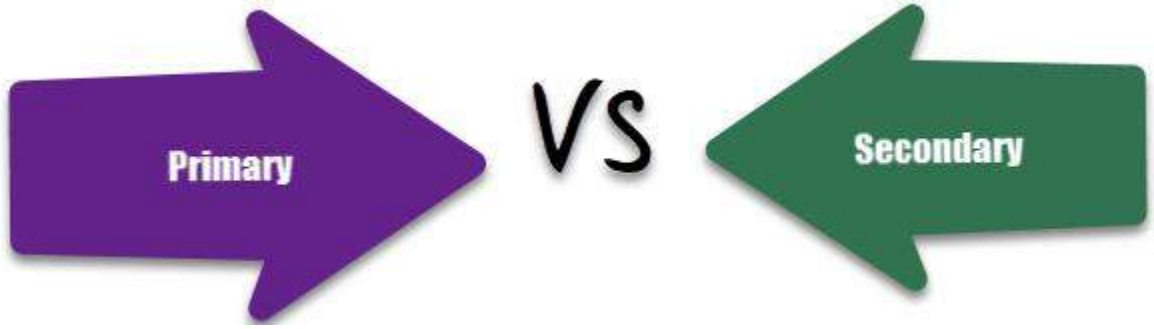
- The computer can't run without primary memory
- It is known as the main memory.
- You can lose data in case power is switched off
- It is also known as volatile memory

- It is a working memory of the computer.
- Primary memory is faster compares to secondary memory.

Characteristic Secondary Memory

- These are magnetic and optical memories
- Secondary memory is known as a backup memory
- It is a non-volatile type of memory
- Data is stored permanently even when the power of the computer is switched off
- It helps store data in a computer
- The machine can run without secondary memory
- Slower than primary memory

Primary Memory Vs Secondary Memory



Parameter	Primary memory	Secondary memory
Nature	The primary memory is categorized as volatile & nonvolatile memories.	The secondary memory is always a non-volatile memory.
Alias	These memories are also called internal memory.	Secondary memory is known as a Backup memory or Additional memory or Auxiliary memory.
Access	Data is directly accessed by the processing unit.	Data cannot be accessed directly by the processor. It is first copied from secondary memory to primary memory. Only then CPU can access it.
Formation	It's a volatile memory meaning data cannot be retained in case of power failure.	It's a non-volatile memory so that that data can be retained even after power failure.
Storage	It holds data or information that is currently being used by the processing unit. Capacity is usually in 16 to 32 GB	It stores a substantial amount of data and information. Capacity is generally from 200GB to terabytes.
Accesses	Primary memory can be accessed by the data bus.	Secondary memory is accessed by I/O channels.

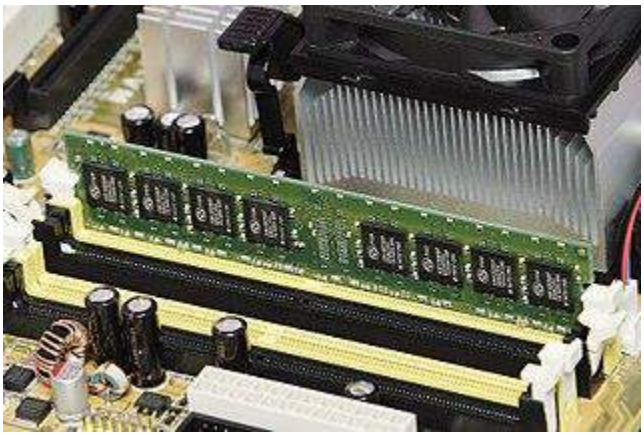
Parameter	Primary memory	Secondary memory
Expense	Primary memory is costlier than secondary memory.	Secondary memory is cheaper than primary memory.

Summary:

- Computer memory is the storage space where data is to be processed, and instructions needed for processing are stored
- Two types of memories are 1) Primary Memory, 2)Secondary Memory
- Primary memory is the main memory of the computer system. Accessing data from primary memory is faster because it is an internal memory of the computer.
- All secondary storage devices which are capable of storing high volume data is referred to as secondary memory
- Types of Primary Memory 1) RAM, 2) ROM
- Types of Secondary Memory 1) Hard Drive, 2) SSD, 3) Flash, 4) Optical Drive, 5) USD Drive, 3) Magnetic Tapes
- The computer can't run without primary memory. You can lose data in case power is switched off
- Data is stored permanently in Secondary Memory even when the power of the computer is switched off
- Primary memory is expensive and is available in limited in size in a computer.
- Secondary memory is cheaper compared to primary memory.

What is memory management in operating system?
In **operating systems**, **memory management** is the function responsible for **managing** the computer's primary **memory**. It tracks when **memory** is freed or unallocated and updates the status. This is distinct from application **memory management**, which is how a process manages the **memory** assigned to it by the **operating system**.

In Operating systems **memory management** is the function responsible for managing the computer's Primary Memory



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

What is memory management in operating systems?

Memory management is the process of controlling and coordinating computer **memory**, assigning portions called blocks to various running programs to optimize overall system performance. **Memory management** resides in hardware, in the OS (operating system), and in programs and applications.

What is the role of memory management in operating systems?

- It is responsible for **managing** the computer's primary **memory**.
- **It** keeps track of the status of each **memory** location, either allocated or free.
- When **memory** is allocated it determines which **memory** locations will be assigned.

What is paged memory management?

- **Paging** is a **memory management** scheme by which a computer stores and retrieves data from secondary storage for use in main **memory**.
- The operating system retrieves data from secondary storage in same-size blocks called pages.

Memory management techniques

The memory management techniques is divided into two parts...

- **Uniprogramming:**

  In the uniprogramming technique, the RAM is divided into two parts one part is for the resigning the operating system and other portion is for the user process. Here the fence register is used which contain the last address of the operating system parts. The operating system will compare the user data addresses with the fence register and if it is different that means the user is not entering in the OS area. Fence register is also called boundary register and is used to prevent a user from entering in the operating system area. Here the CPU utilization is very poor and hence multiprogramming is used.

  Multiprogramming:

  - In the multiprogramming, multiple users can share the memory simultaneously.
  - By multiprogramming we mean there will be more than one process in the main memory

- if the running process wants to wait for an event like I/O then instead of sitting ideal CPU will make a context switch and will pick another process.

Contiguous and non-contiguous memory allocation

Memory is a large array of bytes, where each byte has its own address. The memory allocation can be classified into two methods:

- ✓ **contiguous memory allocation** and
- ✓ **non-contiguous memory allocation**.

The major difference between Contiguous and Noncontiguous memory allocation is that the **contiguous memory allocation** assigns the consecutive blocks of memory to a process requesting for memory whereas, the **noncontiguous memory allocation** assigns the separate memory blocks at the different location in memory space in a nonconsecutive manner to a process requesting for memory.

Frequently, a file stored on disk can become fragmented, which means that it is stored on **non-contiguous** sectors.

Differences:

Comparison Chart

Basis the Comparison	Contiguous Memory Allocation	Noncontiguous Memory Allocation
Basic	Allocates consecutive blocks of memory to a process.	Allocates separate blocks of memory to a process.

Basis the Comparison	Contiguous Memory Allocation	Noncontiguous Memory Allocation
Overheads	**Contiguous** memory allocation **does not** have the overhead of address translation while execution of a process.	**Noncontiguous** memory allocation **has overhead** of address translation while execution of a process.
Execution rate	A process executes faster in contiguous memory allocation	A process executes quite slower comparatively in noncontiguous memory allocation.
Solution	The memory space must be divided into the fixed-sized partition and each partition is allocated to a single process only.	Divide the process into several blocks and place them in different parts of the memory according to the availability of memory space available.
Table	A table is maintained by operating system which maintains the list of available and occupied partition in the memory space	A table has to be maintained for each process that carries the base addresses of each block which has been acquired by a process in memory.

Definition of Contiguous Memory Allocation

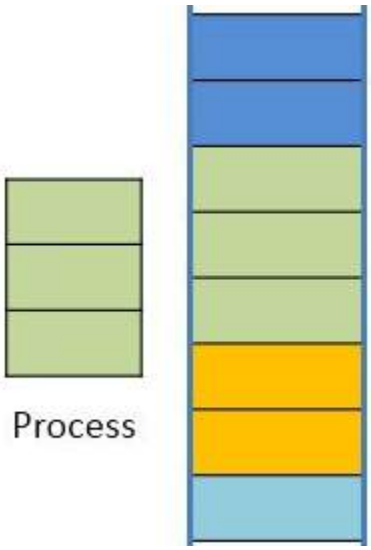
The operating system and the user's processes both must be accommodated in the main memory. Hence the main memory is **divided into two** partitions:

- ✓ at one partition the operating system resides and
- ✓ at other the user processes reside.

Usually, several user processes must reside in the memory at the same time, and therefore, it is important to consider the allocation of memory to the processes.

The Contiguous memory allocation is one of the methods of memory allocation.

- ✓ When a process requests for the memory, a **single contiguous section of memory blocks** is assigned to the process according to its requirement.



The contiguous memory allocation can be achieved by dividing the memory into the fixed-sized **partition** and allocate each partition to a single process only.

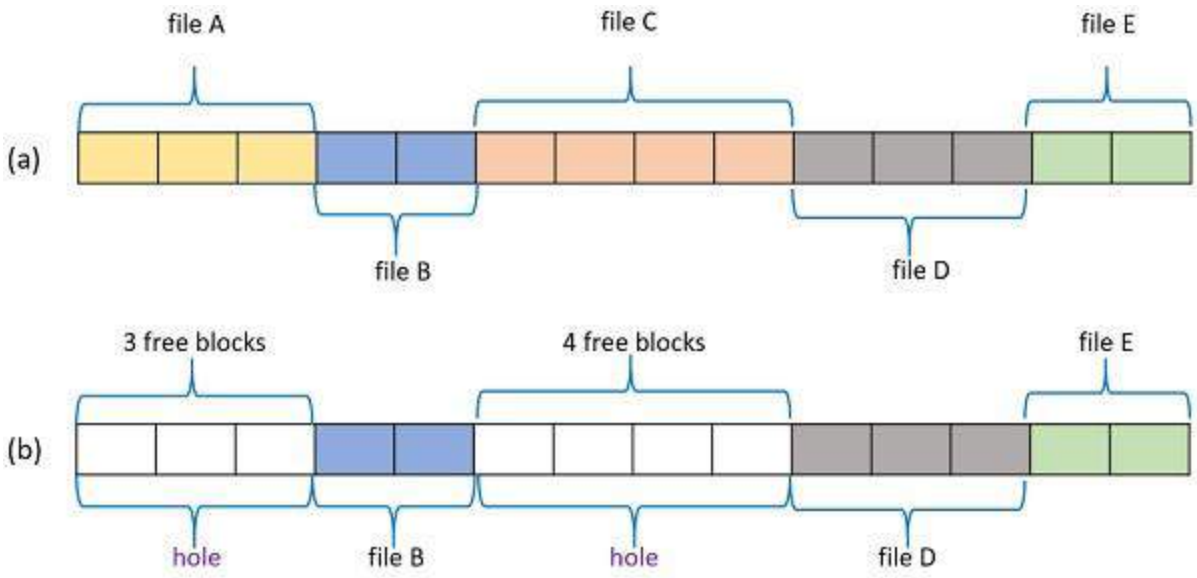
The contiguous memory allocation also leads to the **internal fragmentation**. If a fixed sized memory block allocated to a process is slightly larger than its requirement then the left over memory space in the block is called internal fragmentation.

When the process residing in the partition terminates, the partition becomes available for the process.

Also, the operating system maintains a **table** which indicates, which partition of the memory is free and which is occupied by the processes.

The contiguous memory allocation fastens the execution of a process by reducing the overheads of address translation.

In Contiguous memory allocation, when the process arrives from the ready queue to the main memory for execution, the contiguous memory blocks are allocated to the process according to its requirement. Now, to allocate the **contiguous** space to user processes, the memory can be divide either in the fixed-sized partition or in the variable-sized partition.



(a) Contiguous memory allocation of 5 files
(b) When the file A and C terminates and release the memory creating hole

Fixed-Sized Partition: In the fixed-sized partition, the memory is divided into fixed-sized blocks and each block contains exactly one process. But, the fixed-sized partition will limit the degree of multiprogramming as the number of the partition will decide the number of processes.

Variable-Size Partition: In the variable size partition method, the operating system maintains a table that contains the information about all memory parts that are **occupied** by the processes and all memory parts that are still **available** for the processes.

How holes are created in the memory?

Initially, the whole memory space is available for the user processes as a large block, a **hole**. Eventually, when the processes arrive in the memory, executes, terminates and leaves the memory you will see the set of holes of variable sizes.

In the figure above, you can see that when file A and file C release the memory allocated to them, creates the holes in the memory of variable size.

In the **variable size partition** method, the operating system analyses the memory requirement of the process and see whether it has a memory block of the required size. If it finds the match, then it allocates that memory block to the process. If not,

then it searches the ready queue for the process that has a smaller memory requirement.

The operating system allocates the memory to the process until it cannot satisfy the memory requirement of the next process in the ready queue. It stops allocating memory to the process if it does not have a memory block (**hole**) that is large enough to hold that process.

If the memory block (**hole**) is too **large** for the process it gets **spilt** into two parts. One part of the memory block is allocated to the arrived process and the other part is returned to the set of holes. When a process terminates and releases the memory allocated to it, the released memory is then placed back to the set of holes. The **two holes that are adjacent** to each other, in the set of holes, are merged to form one **large hole**.

Now, at this point, the operating system checks whether this newly formed free large hole is able to satisfy the memory requirement of any process waiting in the ready queue. And the process goes on.

Memory Management

In the above section, we have seen, how the operating system allocates the contiguous memory to the processes. Here, we will see how the operating system selects a free hole from the set of holes?

The operating system uses either the block allocation list or the bit map to select the hole from the set of holes.

Block Allocation List

Block allocation list maintains two tables. One table contains the entries of the blocks that are allocated to the various files. The other table contains the entries of the holes that are free and can be allocated to the process in the waiting queue.

Now, as we have entries of free blocks which one must be chosen can be decided using either of these strategies: first-fit, best-fit, worst-fit strategies.

1. First-fit

Here, the searching starts either at the beginning of the table or from where the previous first-fit search has ended. While searching, the first hole that is found to be large enough for a process to accommodate is selected.

2. Best-fit

This method needs the list of free holes to be sorted according to their size. Then the smallest hole that is large enough for the process to accommodate is selected from the list of free holes. This strategy reduces the wastage of memory as it does not allocate a hole of larger size which leaves some amount of memory even after the process accommodates the space.

3. Worst-fit

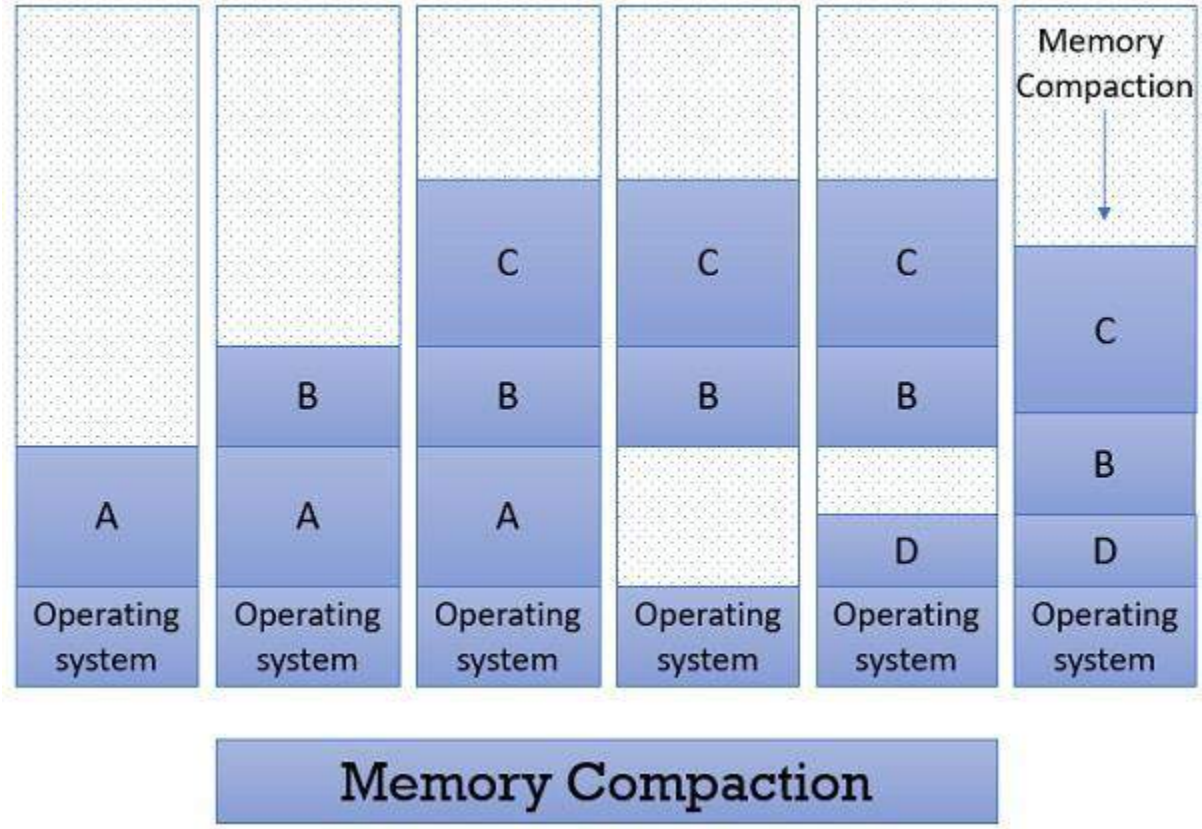
This method requires the entire list of free holes to be sorted. Here, again the largest hole among the free holes is selected. This strategy leaves the largest leftover hole which may be useful for the other process.

Fragmentation

Fragmentation can either be external fragmentation or internal fragmentation. **External fragmentation** is when the free memory blocks available in memory are too small and even non-contiguous. Whereas, the **internal fragmentation** occurs when the process does not fully utilize the memory allocated to it.

The solution to the problem of external fragmentation is **memory compaction**. Here, all the memory contents are shuffled to the one area of memory thereby creating a large block of memory which can be allocated to one or more new processes.

As in the figure below, you can see at the last process C, B, D are moved downwards to make a large hole.



Advantages and Disadvantages

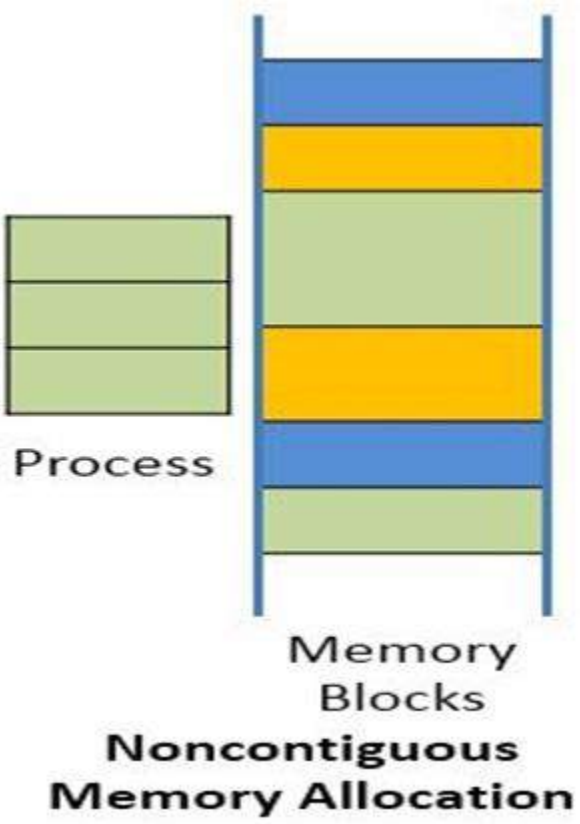
The main **disadvantage** of contiguous memory allocation is **memory wastage** and **inflexibility**. As the memory is allocated to a file or a process keeping in mind that it will grow during the run. But until a process or a file grows many blocks allocated to it remains unutilized. And they even they cannot be allocated to the other process leading to wastage of memory.

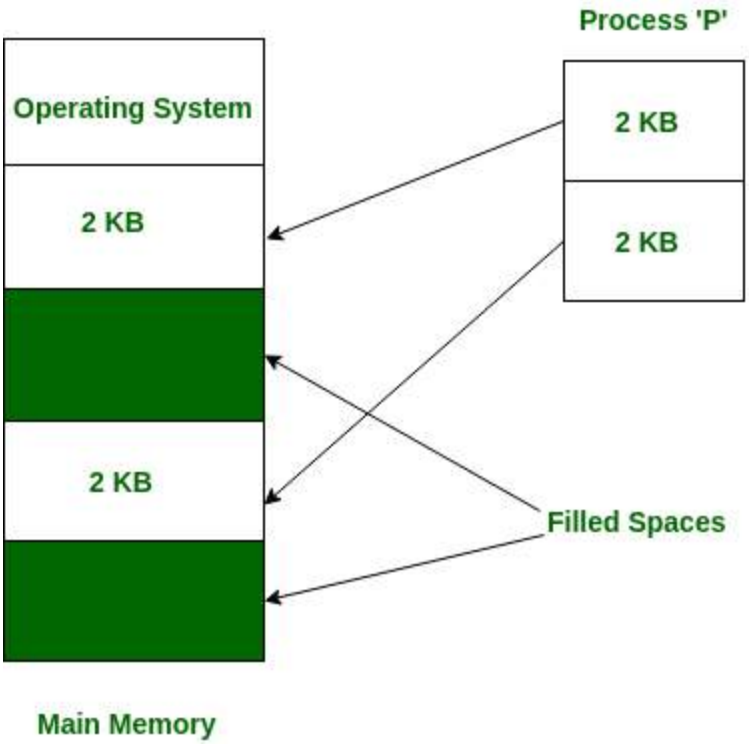
In case, the process or the file grows beyond the expectation i.e. beyond the allocated memory block, then it will abort with the message "No disk space" leading to inflexibility.

The **advantage** of contiguous memory allocation is it increases the processing speed. As the operating system uses the buffered I/O and reads the process memory blocks consecutively it reduces the head movements. This speed ups the processing.

Non-contiguous memory allocation

In the **non-contiguous memory allocation** the available free memory space are scattered here and there and all the free memory space is not at one place. So this is time-consuming. In the **non-contiguous memory allocation**, a process will acquire the memory space but it is not at one place it is at the different locations according to the process requirement. This technique of **non-contiguous memory allocation** reduces the wastage of memory which leads to internal and external fragmentation. This utilizes all the free memory space which is created by a different process.





Non-contiguous memory allocation is of different types,

1. Paging
2. Segmentation
3. Segmentation with paging

i) Paging

A non-contiguous policy with a fixed size partition is called paging. A computer can address more memory than the amount of physically installed on the system. This extra memory is actually called virtual memory. Paging technique is very important in implementing virtual memory. Secondary memory is divided into equal size partition (fixed) called pages. Every process will have a separate page table. The entries in the page table are the number of pages a process. At each entry either we have an invalid pointer which means the page is not in main memory or we will get the corresponding frame number. When the frame number is combined with instruction of set D than we will get the corresponding physical address. Size of a page table is generally very large so cannot be accommodated

inside the PCB, therefore, PCB contains a register value PTBR(page table base register) which leads to the page table.

Advantages: It is independent of external fragmentation.

Disadvantages:

1. It makes the translation very slow as main memory access two times.
2. A page table is a burden over the system which occupies considerable space.

ii) Segmentation

Segmentation is a programmer view of the memory where instead of dividing a process into equal size partition we divided according to program into partition called segments. The translation is the same as paging but paging segmentation is independent of internal fragmentation but suffers from external fragmentation. Reason of external fragmentation is program can be divided into segments but segment must be contiguous in nature.

iii) Segmentation with paging

In segmentation with paging, we take advantages of both segmentation as well as paging. It is a kind of multilevel paging but in multilevel paging, we divide a page table into equal size partition but here in segmentation with paging, we divide it according to segments. All the properties are the same as that of paging because segments are divided into pages.

Conclusion:

Contiguous memory allocation does not create any overheads and fastens the execution speed of the process but **increases memory wastage**. In turn noncontiguous memory allocation creates overheads of address translation, reduces

execution speed of a process but, **increases memory utilization**. So there are pros and cons of both allocation methods.

Difference between Logical and Physical Address in Operating System

Address uniquely identifies a location in the memory. We have two types of addresses that are logical address and physical address. The logical address is a virtual address and can be viewed by the user. The user can't view the physical address directly. The logical address is used like a reference, to access the physical address. The fundamental difference between logical and physical address is that **logical address** is generated by CPU during a program execution whereas, the **physical address** refers to a location in the memory unit.

Comparison Chart

Basis for Comparison	Logical Address	Physical Address
Basic	It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Address Space	Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
Visibility	The user can view the logical address of a program.	The user can never view physical address of program
Access	The user uses the logical address to access the physical address.	The user can not directly access physical address.
Generation	The Logical Address is generated by the CPU	Physical Address is Computed by MMU

Definition of Logical Address

Address generated by **CPU** while a program is running is referred as **Logical Address**. The logical address is virtual as it does not exist physically. Hence, it is also called as **Virtual Address**. This address is used as a reference to access the physical memory location. The set of all logical addresses generated by a programs perspective is called **Logical Address Space**.

The logical address is mapped to its corresponding physical address by a hardware device called **Memory-Management Unit**. The address-binding methods used by MMU generates **identical** logical and physical address during **compile time** and **load time**. However, while **run-time** the address-binding methods generate **different** logical and physical address.

Definition of Physical Address

Physical Address identifies a physical location in a memory. MMU (**Memory-Management Unit)** computes the physical address for the corresponding logical address. MMU also uses logical address computing physical address. The user never deals with the physical address. Instead, the physical address is accessed by its corresponding logical address by the user. 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. Hence, the logical address must be mapped to the physical address before they are used.

The logical address is mapped to the physical address using a hardware called **Memory-Management Unit**. The set of all physical addresses corresponding to the logical addresses in a Logical address space is called **Physical Address Space**.

Key Differences Between Logical and Physical Address in OS

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program. On the other hand, the physical address is a location that exists in the memory unit.
2. The set of all logical addresses generated by CPU for a program is called Logical Address Space. However, the set of all physical address mapped to corresponding logical addresses is referred as Physical Address Space.
3. The logical address is also called virtual address as the logical address does not exist physically in the memory unit. The physical address is a location in the memory unit that can be accessed physically.
4. Identical logical address and physical address are generated by Compile-time and Load time address binding methods.
5. The logical and physical address generated while run-time address binding method differs from each other.
6. The logical address is generated by the CPU while program is running whereas, the physical address is computed by the MMU (Memory Management Unit).

Conclusion:

The logical address is a reference used to access physical address. The user can access physical address in the memory unit using this logical address.

Difference Between Virtual and Cache Memory in OS

Memory is a hardware device that is used to store the information either temporary or permanently. In this article, I have discussed the differences between virtual and cache memory. A **Cache memory** is a high-speed memory which is used to reduce the access time for data. On the other hands, **Virtual memory** is not exactly a physical memory it is a technique which extends the capacity of the main memory beyond its limit.

The major difference between virtual memory and the cache memory is that a **virtual memory** allows a user to execute programs that are larger than the main memory whereas, **cache memory** allows the quicker access to the data which has been recently used. We will discuss some more differences with the help of comparison chart shown below.

Comparison Chart

Basis for Comparison	Virtual Memory	Cached Memory
Basic	Virtual memory extends the capacity of main memory for the user.	Cache memory fastens the data accessing speed of CPU.
Nature	Virtual memory is technique.	Cache memory is a storage unit.
Function	Virtual memory allows execution of the program that is larger than the main memory.	Cache memory stores the copies of original data that has been recently used.
Memory management	Virtual memory is managed by the operating system.	Cache memory is fully managed by the hardware.
Size	Virtual memory is far larger than cached memory.	Cache memory has bounded size.
Mapping	Virtual memory requires mapping structures to map virtual address to physical address.	No mapping structures are required as such in a cache memory.

Definition of Virtual Memory

Virtual memory is not exactly a physical memory of a computer instead it's a **technique** that allows the execution of a **large program** that may **not** be **completely placed in the main memory**. It enables the programmer to execute the programs larger than the main memory.

Now let us understand how does the virtual memory works? The program has its virtual memory address which is divided into a number of **pages**. The main memory is also divided into a number of **pages**. Now, as we can see the virtual address of a program is larger than the available main memory. So memory map is used to map the virtual address to the main memory.

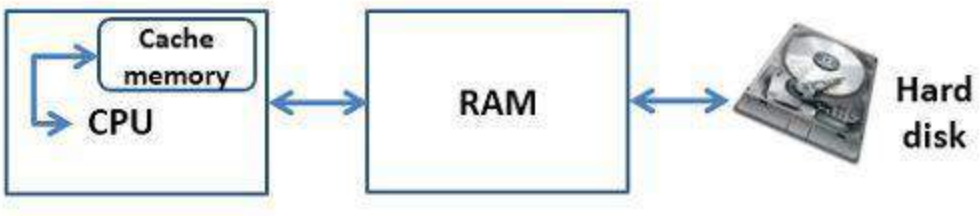
The benefits of virtual memory are:

- The programs are no longer constrained by the limit of main memory.
- Virtual memory increases the degree of multiprogramming.
- Increases CPU utilization.
- The less I/O unit will require to load or to swap programs in the memory.

But there is a **drawback** of virtual memory, placing more pages of a program in hard disk will **slow** down the **performance** as accessing the data from hard disk takes more time in comparison to accessing data from main memory.

Definition of Cache Memory

Unlike virtual memory, **Cache** is a **storage device** implemented on the **processor** itself. It carries the copies of original data that has been accessed recently. The original data may be placed in the main memory or a secondary memory. The cache memory **fastens** the accessing speed of data.



We can say that the accessing speed of CPU is **limited** to the accessing speed of **main memory**. Whenever a program is to be executed by the processor, it fetches it from main memory. If **a copy** of the program is already **present** in the **cache** implemented on the processor. The process would be able to access that data faster which will result in faster execution.

Key Differences Between Virtual and Cache Memory

1. Virtual memory **extends** the capacity of main memory virtually for the user. However, the cache memory makes the accessing of data **faster** for CPU.
2. Cache is a memory **storage unit** whereas as the Virtual memory is a **technique**.
3. Virtual memory enables the executions of the program that **larger** than the main memory. On the other hands, cache memory stores the **copies** of original data that were used recently.
4. Virtual memory management is done by the **operating system**. On the other hands, cache memory management is done by the **hardware**.
5. Virtual memory is far **larger** than the cached memory in size.
6. Virtual memory technique requires the **mapping structures** to map virtual address to physical address whereas, cache memory **does not** require any mapping structures.

Conclusion:

The Virtual memory is a technique to expand the capacity of main memory virtually for the users. The cache memory is a storage unit that stores the recently accessed data which enables the CPU to access it faster.

Difference between Internal and External fragmentation

Whenever a process is loaded or removed from the physical memory block, it creates a small hole in memory space which is called fragment. Due to fragmentation, the system fails in allocating the contiguous memory space to a process even though it have the requested amount of memory but, in a non-contiguous manner. The fragmentation is further classified into two categories Internal and External Fragmentation.

Both the internal and external classification affects data accessing speed of the system. They have a basic difference between them i.e. **Internal fragmentation** occurs when fixed sized memory blocks are allocated to the process without concerning about the size of the process, and **External fragmentation** occurs when the processes are allocated memory dynamically. Let us move further and discuss the differences, reasons, solutions behind internal and external fragmentation with the help of comparison chart shown below.

Definition of Internal Fragmentation

Internal fragmentation occurs when the memory is divided into **fixed sized blocks**. Whenever a process request for the memory, the fixed sized block is allocated to the process. In case the memory assigned to the process is somewhat larger than the memory requested, then the difference between assigned and requested memory is the **Internal fragmentation**.

This leftover space inside the fixed sized block cannot be allocated to any process as it would not be sufficient to satisfy the request of memory by the process. The memory space is partitioned into the fixed-sized blocks of 18,464 bytes. Let us say a process request for 18,460 bytes and partitioned fixed-sized block of 18,464 bytes is allocated to the process. The result is 4 bytes of 18,464 bytes remained empty which is the internal fragmentation.

The overhead of keeping track of the internals hole created due to internal fragmentation is substantially more than the number of internal holes. The problem of internal fragmentation can be solved by **partitioning the memory into the variable sized block** and assign the best-sized block to a process requesting for the memory. Still, it will not totally eliminate the problem of internal fragmentation but will reduce it to some extent.

Definition of External Fragmentation

External fragmentation occurs when there is a sufficient amount of space in the memory to satisfy the memory request of a process. But the process's memory request cannot be satisfied as the memory available is in a non-contiguous manner. Either you apply first-fit or best-fit memory allocation strategy it will cause external fragmentation.

When a process is loaded and removed from the memory the free space creates the hole in the memory space, and there are many such holes in the memory space, this is called External fragmentation. Although the first fit and best fit can affect the amount of external fragmentation, it cannot be totally eliminated. **Compaction** may be the solution for external fragmentation.

Compaction algorithm shuffles all memory contents to one side and frees one large block of memory. But compaction algorithm is expensive. There is an alternative solution to solve external fragmentation issue which will allow a process to acquire physical memory in a non-contiguous manner. The techniques to achieve this solution are paging and segmentation.

Key Differences between Internal and External fragmentation

1. The basic reason behind the occurrences of internal and external fragmentation is that internal fragmentation occurs when memory is partitioned into **fixed-sized blocks** whereas external fragmentation occurs when memory is partitioned into **variable size blocks**.
2. When the memory block allotted to the process comes out to be slightly larger than requested memory, then the free space left in the allotted memory block causes internal fragmentation. On the other hands, when the process is removed from the memory it creates free space causing a hole in the memory which is called external fragmentation.
3. The problem of internal fragmentation can be solved by partitioning the memory into variable sized blocks and assign the best fit block to the requesting process. However, the solution for external fragmentation is compaction, but it is expensive to implement, so the processes must be allowed to acquire physical memory in a non-contiguous manner, to achieve this the technique of paging and segmentation is introduced.

Conclusion:

The problem of internal fragmentation can be reduced, but it cannot be totally eliminated. The paging and segmentation help in utilizing the space freed due to

external fragmentation by allowing a process to occupy the memory in a non-contiguous manner.

Comparison Chart

Basis for Comparison	Internal Fragmentation	External Fragmentation
Basic	It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
Occurrence	When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution	The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

Comparison Chart

Basis of Comparison	Paging	Segmentation
Basic	Paging allows the memory address space of a process to be noncontiguous.	Swapping allows multiple programs to run parallel in the operating system.

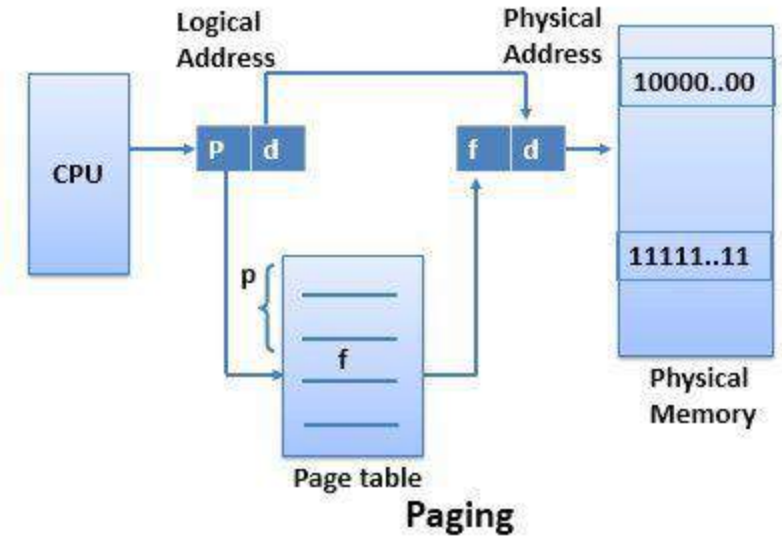
Basis of Comparison	Paging	Segmentation
Flexibility	Paging is more flexible as only pages of a process are moved.	Swapping is less flexible as it moves entire process back and forth between main memory and back store.
Multiprogramming	Paging allows more processes to reside in main memory	Compared to paging swapping allows less processes to reside in main memory.

Definition of Paging

Paging is a memory management scheme, which allocates a **noncontiguous address space** to a process. Now, when a process's physical address can be non-contiguous the problem of **external fragmentation** would not arise.

Paging is implemented by breaking the **main memory** into fixed-sized blocks that are called **frames**. The **logical memory of a process** is broken into the same fixed-sized blocks called **pages**. The page size and frame size is defined by the hardware. As we know, the process is to be placed in main memory for execution. So, when a process is to be executed, the pages of the process from the source i.e. back store are loaded into any available frames in main memory.

How paging is implemented. CPU generates the logical address for a process which consists of two parts that are **page number** and the **page offset**. The page number is used as an **index** in the **page table**.

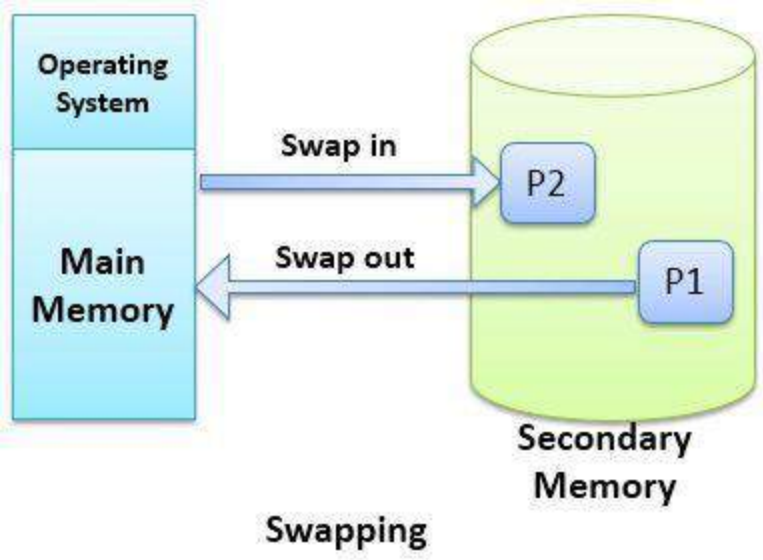


The page table contains the **base address** of each page that loaded in main memory. This base address is combined with page offset to generate the address of the page in main memory.

Every operating system has its own way of storing page table. Most of the operating system has a separate page table for each process.

Definition of Swapping

For execution, each process must be placed in the main memory. When we need to execute a process, and the main memory is entirely full, then the **memory manager swaps** a process from main memory to backing store by evacuating the place for the other processes to execute. The memory manager swaps the processes so frequently that there is always a process in main memory ready for execution.



Swapping

Due to **address binding** methods, the process that is swapped out of main memory occupies same address space when it is swapped back to the main memory if the binding is done at the assembly or load time. If the binding is done at execution time, the process can occupy any available address space in main memory as addresses are computed at the execution time.

Although the performance is affected by swapping, it helps in running **multiple processes in parallel**.

Key Differences between Paging and Swapping in OS

1. The basic difference between paging and swapping is that paging avoids **external fragmentation** by allowing the physical address space of a process to be noncontiguous whereas, swapping allows **multiprogramming**.
2. Paging would transfer pages of a process back and forth between main memory, and secondary memory hence paging is flexible. However swapping swaps entire process back and forth between the main and secondary memory and hence swapping is less flexible.
3. Paging can allow more processes to be in main memory than the swapping.

Conclusion:

Paging avoids external fragmentation as utilizes the non-contiguous address spaces in the main memory. Swapping could be added to the CPU scheduling algorithm where process frequently needs to be in and out of main memory.