Introduction to OS Notes

An Introduction To The Operating System

What is an Operating System?

An **operating system** is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner.

Operating system objectives:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Design terms

- Easy (ease of use)
- Personal computers
- Efficiency (allocation of appropriate resources)
- High performance computers
- Energy saving
- Minor user interruptions
- Device is installed

Tasks offered by an Operating System

- **Program implementation**

Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.

- **I/O Performance**

I/O means any file or any specific I/O device. Program may require any I/O device while running. So the operating system must provide the required I/O.

- **File system manipulation**

Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.

- **Communication**

Data transfer between two processes is required for some lime. The both processes are on the one computer or on different computers but connected through computer network Communication may be implemented by two methods: shared memory and message passing.

- **Error detection**

Error may occur in CPU, in I/O device or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

Operating System Types

- **Single process operating system**

A computer system that allows only one user to use the computer at a given time is known as a single-user system. The goals of such systems are maximizing user convenience and responsiveness, instead of maximizing the utilization of the CPU and peripheral devices.

```
Example: Windows, Apple Mac
OS,etc.
```

- **Batch-processing system**

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having the same requirement and groups them into batches. It is the responsibility of the operator to sort jobs with similar needs.

```
Example: Payroll System,Bank
Statements
```

- **Time-Sharing Operating System**

The **Time Shared Operating System** is also known as the Multitasking Operating System.Time-sharing operating systems implements CPU scheduling and multi programming systems which deliver to every user a small piece of operating time.

```
Example: UNIX, Multics, Linux, Windows 2000
server,etc.
```

- **Distributed Operating System**

Distributed operating system allows distributing of entire systems on the couples of center processors, and it serves on the multiple real time products as well as multiple users.

```
Example: Windows Server 2003, Windows Server 2008, Windows Server 2012,
Ubuntu, Linux(Apache Server),etc.
```

- **Real Time Operating System**

Real time systems are used when strict time requirements are placed on the operation of a processor or the flow of data. These are used to control a device in a dedicated application.

```
Example: Scientific experiments, medical imaging systems, industrial
control systems
, weapon systems, robots, air traffic control systems, etc.
```

- **Network Operating System**

Network Operating System is a computer operating system that facilitates to connect and communicate various autonomous computers over a network. An Autonomous computer is an independent computer that has its own local memory, hardware, and O.S. It is self capable to perform operations and processing for a single user. They can either run the same or different O.S.

```
Example: Microsoft Windows Server 2003, Microsoft Windows Server
2008, UNIX, Linux,
```

Mac OS X, Novell NetWare, and BSD, etc

Process Notes

Process

What is a process?

The process is an example of a computer program used. Contains the program code and its current function. Depending on the operating system (OS), a process can be performed with multiple configurations that issue commands simultaneously. Each process has a complete set of variations.

Process Conditions:

- **New State:** This is the state when the process is just created. It is the first state of a process.
- **Ready State:** After the creation of the process, when the process is ready for its execution then it goes into the ready state. In a ready state, the process is ready for its execution by the CPU but it is waiting for its turn to come. There can be more than one process in the ready state.
- **Ready Suspended State:** There can be more than one process in the ready state but due to memory constraint, if the memory is full then some process from the ready state gets placed in the ready suspended state.
- **Running State:** Amongst the process present in the ready state, the CPU chooses one process amongst them by using some CPU scheduling algorithm. The process will now be executed by the CPU and it is in the running state.
- **Waiting or Blocked State:** During the execution of the process, the process might require some I/O operation like writing on file or some more priority process might come. In these situations, the running process will have to go into the waiting or blocked state and the other process will come for its execution. So, the process is waiting for something in the waiting state.
- **Waiting Suspended State:** When the waiting queue of the system becomes full then some of the processes will be sent to the waiting suspended state.
- **Terminated State:** After the complete execution of the process, the process comes into the terminated state and the information related to this process is deleted.

Process Control Block(PCB)

Each process is represented in the operating system by a process control block (PCB) also called a task control block. It contains many pieces of information associated with a specific process, including these:

- **Process:** The state may be new, ready, running, and so on
- **Program counter:** It indicates the address of the next instruction to be executed for this program.
- **CPU registers:** These vary in number and type based on architecture. They include accumulators, stack pointers, general-purpose registers, etc.
- **CPU scheduling:** This includes process priority, pointers to scheduling queues, and any scheduling parameters.
- **Memory-management:** This includes the value of base and limit registers (protection) and page tables, segment tables depending on memory.

- **Accounting:** It includes the amount of CPU and real-time used, account numbers, process numbers, etc
- **I/O status information:** It includes a list of I/O devices allocated to this process, a list of open files, etc

Program vs Process

Process	Program
1. Process is an instance of an executing program.	1. Program contains a set of instructions designed to complete a task.
2. Lifespan of process is less than program.	2. Lifespan of program is longer
3. Process exists for a limited span of time as it gets terminated after the completion of a task.	3. Program exists at a single place and continues to exist until it is deleted.
4. Process is a dynamic entity.	4. Program is a static entity.
5. The process has a high resource requirement, it requires resources such as CPU, memory address, O / O during its lifetime.	5. Program has no resource requirement, it only requires memory space to store commands.

Multiprogramming

A computer running more than one program at a time (like running Excel and Firefox simultaneously).
Multiprogramming increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has one to execute.

Multitasking

Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. Multitasking is a logical extension of multiprogramming.
CPU executes multiple tasks by switching among them.
The switching is very fast.
Requires an interactive (hands-on) computer where the user can directly interact with the computer.

Multiprocessing

Multi

Processing sometimes refers to executing multiple processes (programs) at the same time.
A system can be both multi
programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

Threads

Threads

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are a popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving the illusion that the threads are running in parallel.As each thread has its own independent resource for process execution, multiple processes can be executed parallely by increasing the number of threads.

Types of Thread

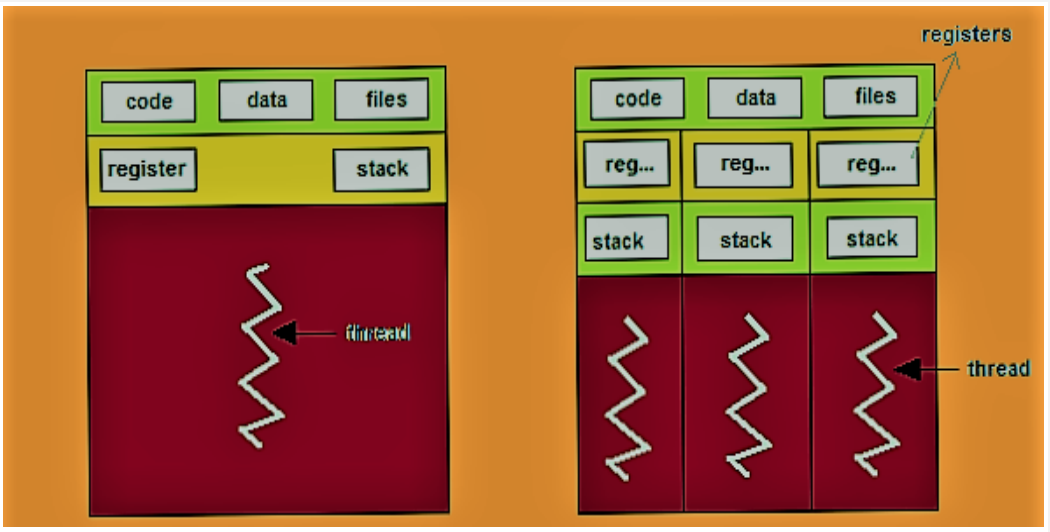
- **User threads** are above the kernel and without kernel support. These are the threads that application programmers use in their programs.
- **Kernel threads** are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Program vs Process vs Thread

Program	Process	Thread
An execution file stored in harddrive.	**An execution file stored in memory.**	**An execution path of part of the process.**
Program contains in the instruction.	**A process is a sequence of instruction.**	**Thread is a single sequence stream within a process.**
One Program contains many processes.	**One Process can contain several threads.**	**One thread can belong to exactly one process.**

Multithreading

Multithreading is a phenomenon of executing multiple threads at the same time. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.



Single threaded Process **Multithreaded Process**

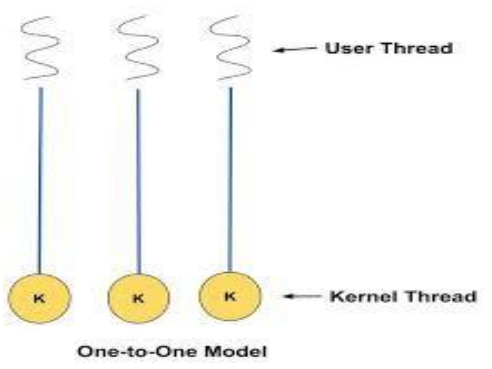
Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies:

Many to One Model

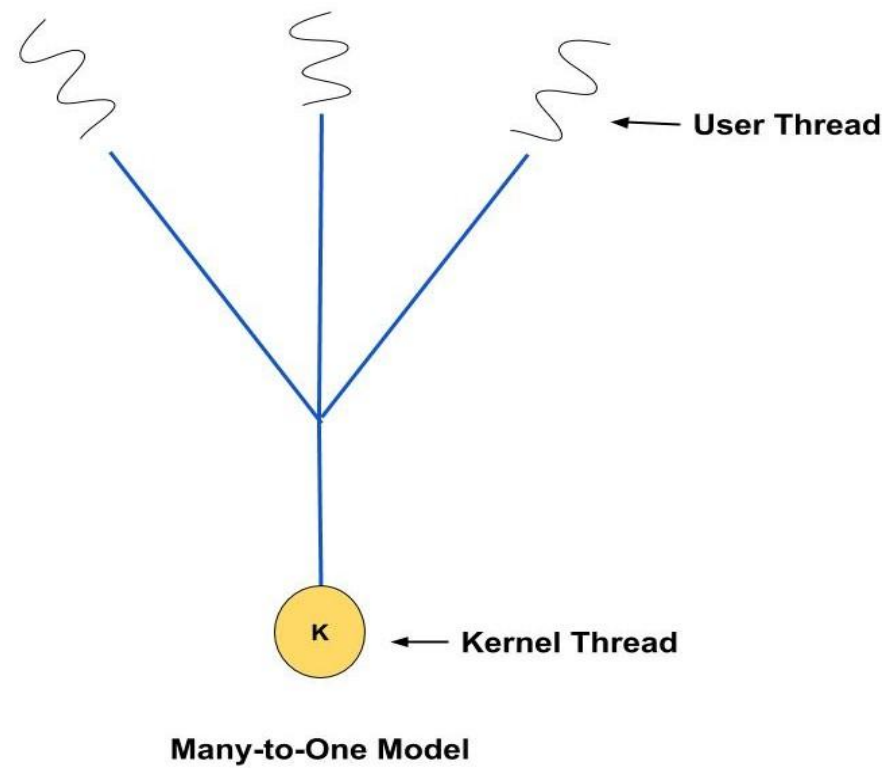
As the name suggests there is many to one relationship between threads. Here, multiple user threads are associated or mapped with one kernel thread. The thread management is done on the user level so it is more efficient.

Example: Initial implementation of Java threads on the Solaris system



Many to One Model

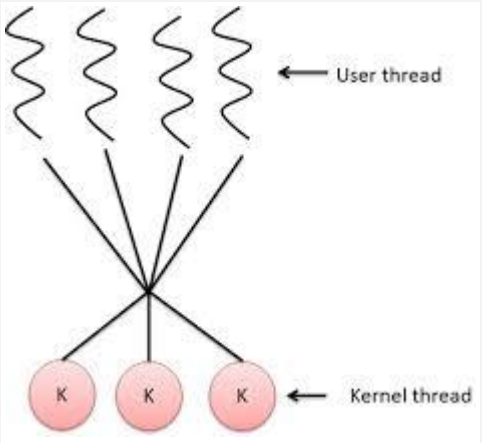
The one to one model creates a separate kernel thread to handle each and every user thread.Most implementations of this model place a limit on how many threads can be created.Linux and Windows from 95 to XP implement the one-to-one model for threads.



Example: Windows NT and the OS/2 threads package

Many to Many Model

The many to many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.Blocking the kernel system calls does not block the entire process.



Many to Many Model

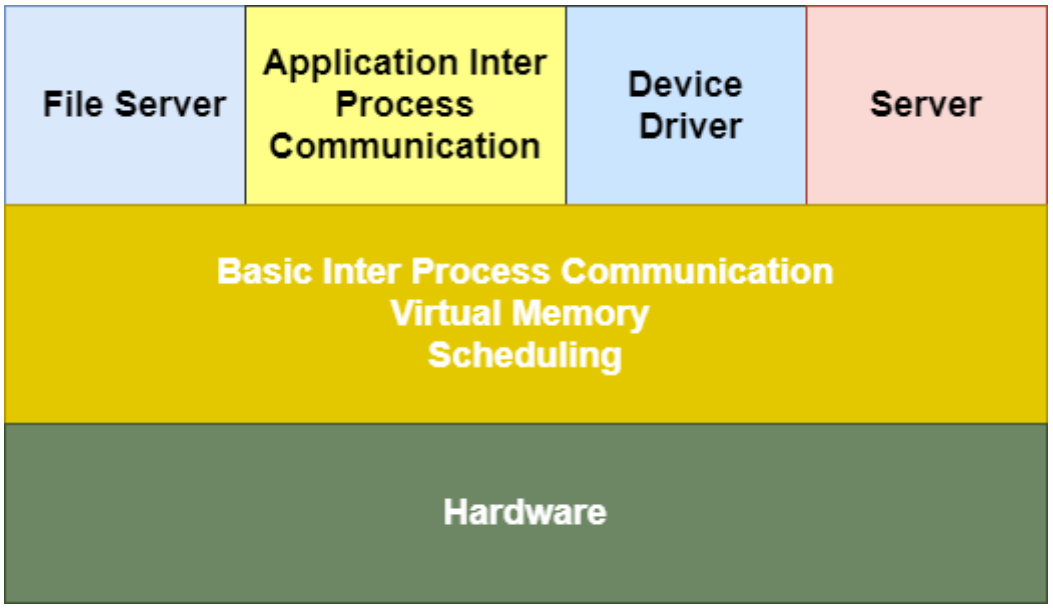
Example: Implementation of Java on an MT operating system.

Kernel

Kernel

A **Kernel** is the central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time. It is the core component of an operating system. Kernel acts as a bridge between applications and data processing performed at hardware level using inter-process communication and system calls.

Kernel Architecture



Micro Kernel Architecture

Types of Kernel

1. Monolithic Kernel: Here, the OS and Kernel both run in the same memory space and are suitable where security is not a significant concern. It results in faster access, but if there is a bug in the device driver, the entire system crashes.

Example: Unix, Linux, Open VMS, XTS-400 etc.

2. Microkernel: It's a stripped-down version of Monolithic Kernel where the Kernel itself can do most of the job, and there is no need for an extra GUI..A microkernel is much smaller in size than a conventional kernel and supports only the core operating system functionalities.

Example: Mach, L4, AmigaOS, Minix, K42 etc.

3. Hybrid Kernel: This Kernel is what we see most. Windows, Apple's macOS. They are a mix of Monolithic Kernel and Microkernel. It moves out drivers but keeps system services inside the Kernel – similar to how drivers are loaded when Windows Starts the bootup process.

Example: Windows NT, Netware, BeOS etc.

4. Nano Kernel: If you need to have a kernel, but its majority of function is set up outside, then this comes into the picture.

Example: EROS etc.

5. Exo Kernel: This kernel only offers process protection and resource handling. However it is mostly used when you are testing out an inhouse project, and you upgrade to a better Kernel type.

Example: Nemesis, ExOS etc.

Shell

A **shell**, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

System calls

A system call is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.User programs typically do not have permission to perform operations like accessing I/O devices and communicating with other programs.
A user program invokes system calls when it requires such services.
System calls provide an interface between a program and the operating system.
System calls are of different types.

Example – fork, exec, getpid, getppid, wait, exit.

Operation mode

1. User Mode
When the computer system runs user applications like creating a text document or using any application program, then the system is in the user mode. When the user application requests for a service from the operating system or an interrupt occurs or system call, then there will be a transition from user to kernel mode to fulfill the requests.
To switch from kernel mode to user mode, mode bit should be 1.

2. Kernel Mode
When the system boots, hardware starts in kernel mode and when the operating system is loaded, it starts user application in user mode. To provide protection to the hardware, we have privileged instructions which execute only in kernel mode. If a user attempts to run privileged instruction in user mode then it will treat the instruction as illegal and traps to OS.
To switch from user mode to kernel mode mode bit should be 0.

Process Scheduling

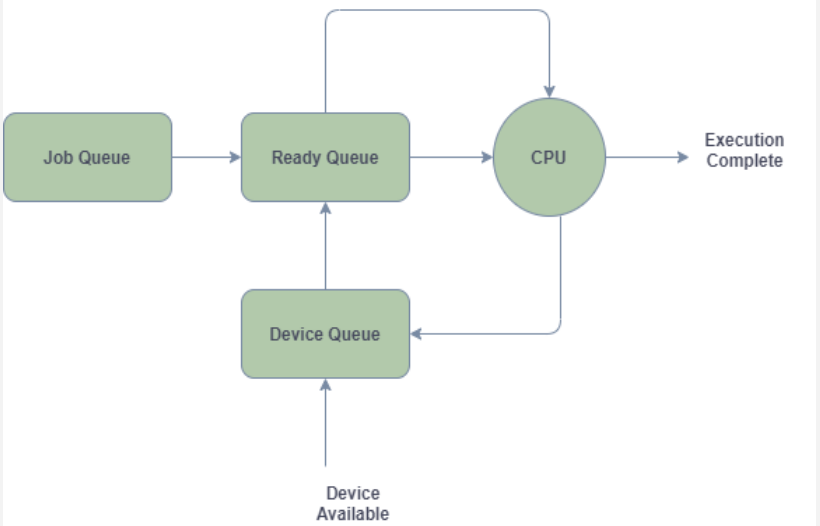
Process Scheduling

The process scheduling is the job of the process manager that handles the removal of the current running process from the CPU and the selection of another process on the basis of a particular approach.It is an essential part of a multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Scheduling Queue

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

Queues are of three types:
Job Queue: As a process enters the system, it is put in a job queue that contains all the processes in the system.
Ready queue: The processes that are residing in the memory and are ready for execution are kept in the ready queue.The ready queue is implemented as a linked list of PCBs with a header containing pointers to the first and the last PCBs.
Device queue: The list of the processes waiting for a particular i/o device is called a device queue.



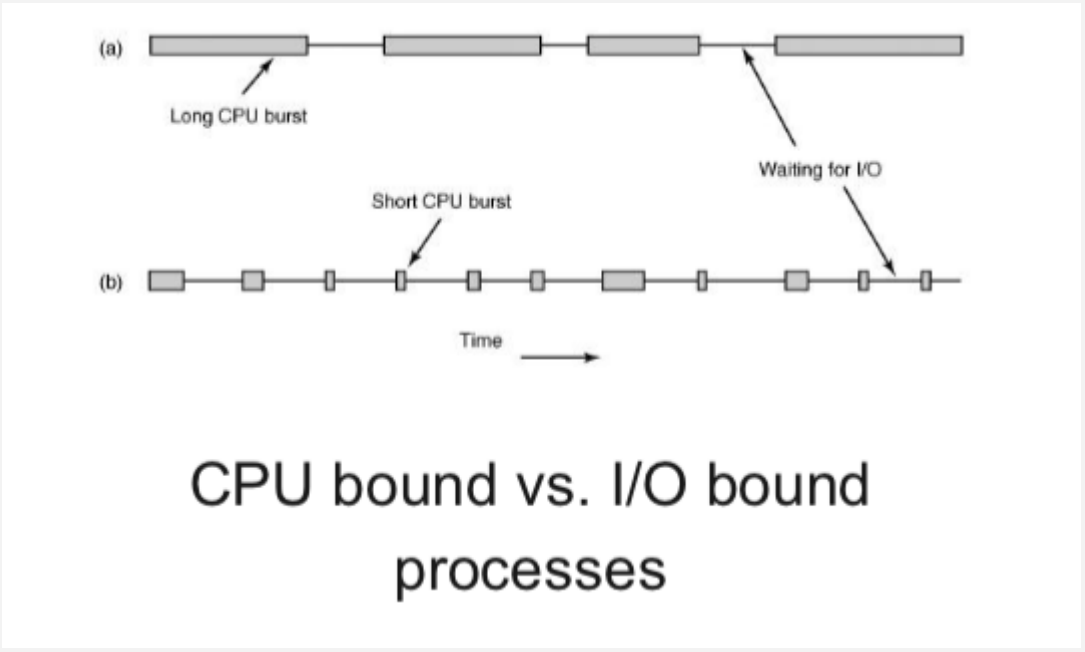
CPU Bound Process

- CPU Bound processes are those that use algorithms with a large number of statistics.
- They can be expected to hold the CPU as long as the editor allows it.
- Programs such as simulations may be tied to the CPU for most of the process life.
- Users do not expect immediate response from a computer when using CPU-bound programs.
- They should be given the most important scheduled schedule.

I/O Bound Process

- The most awaiting processes for the completion of input or output (I / O) are I / O Bound.
- Interactive processes, such as office applications especially I / O are binding throughout the life of the process. Some processes may be bound by I / O in just a few short periods.

- The expected running time of bound I / O processes means that the process will not last very long.
- This should be provided very early by the editor.

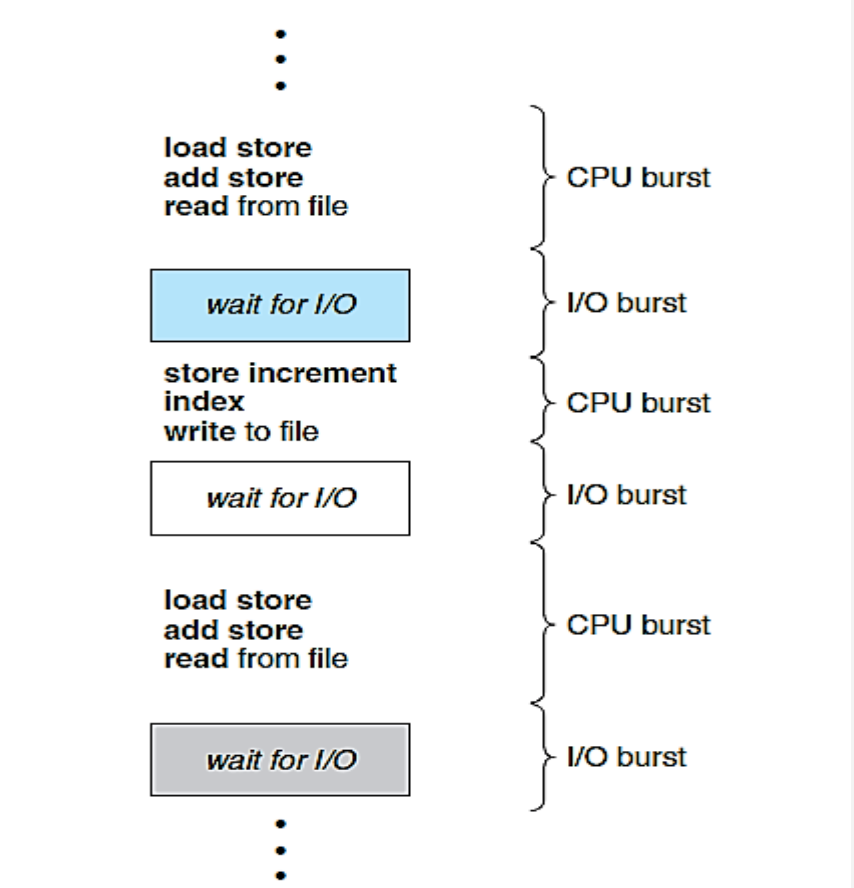


Context Switching

- Context switching is done to switch between processes.
- Switching the CPU to another process requires saving the state of the current process and reloading the state of another process. States are saved into and reloaded from PCBs.
- Context-switch time is a pure overhead as the system does not do any useful work during a control switch.
- Context-switch time depends highly on the hardware.
- Context switching is faster on RISC processors with overlapped register windows.

CPU-I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on.
- Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.



Alternate Sequence of CPU and I/O Burst

CPU Scheduler

When the CPU turns out to be idle then the operating system must select one of the essential processes in the ready queue to be executed. Short term scheduler is used for the selection process. There are several algorithms that are used for ready queue such as FIFO, LIFO priority queue and so on. Generally, PCB is used to keep the record of queues.
CPU scheduling decisions may take place when a process:
1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

There are two types of scheduling:

Preemptive scheduling
In preemptive scheduling, a process can be forced to leave the CPU and switch to the ready queue. Example – Unix, Linux, Windows 95 and higher.

Non-preemptive scheduling
In non-preemptive scheduling or cooperating scheduling, a process keeps the CPU until it terminates or switches to the waiting state. Some machines support non-preemptive scheduling only. Example – Window 3.1x.

Dispatcher

A dispatcher is the module of the operating system that gives control of the CPU to the process selected by the CPU scheduler.
Steps :

- switching context
- switching to user mode
- jumping to the proper location in the user program

Dispatch latency: time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

CPU utilization: keep the CPU as busy as possible
Throughput: No. of processes that complete their execution per time unit
Turnaround time: amount of time to execute a particular process
Waiting time:amount of time a process has been waiting in the ready queue
Response time: amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithms

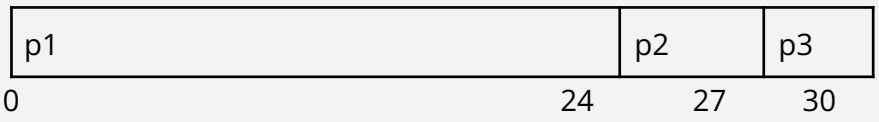
First-come first-served (FCFS) scheduling

The process that asks for the CPU first is given to the CPU first. The implementation of FCFS policy is easily handled with FIFO queue. The average waiting time under the FCFS policy, however, is often quite long.
It is non-preemptive.
It has a high average waiting time.

Example:

Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:



Average waiting time = (0+24+27) / 3 = 17 ms
Average Turnaround time = (24+27+30) / 3 = 27 ms

The FCFS algorithm is particularly troublesome for time – sharing systems, where it is important that each user get a share of the cpu at regular intervals.

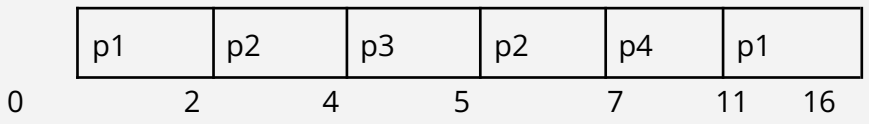
Shortest-Job-First Scheduling

This algorithm is associated with each process length of the next CPU storage. When the CPU is available, it is given a process with the next minimum CPU bust. If the two processes are the similar length for the next CPU explosion, the FCFS configuration is used to break the tire.
If a new process arrives with CPU burst length less than remaining time of the current executing process, preempt,his scheme is known as the Shortest-Remaining-Time-First (SRTF).
SJF is optimal – gives a minimum average waiting time for a given set of processes.

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Gantt Chart:



SJF (preemptive)
Average waiting time = (9 + 1 + 0 +2)/4 = 3 ms

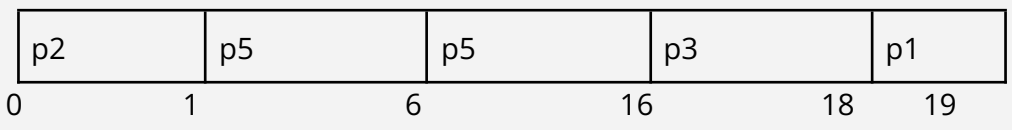
Priority Scheduling

- CPU is allocated to the particular process with the highest priority .
- Priority range be 0 to 7 (say), with 0 representing the highest or the lowest priority
- Priority may depend on internal factors (time limit, memory requirement, number of open files, etc.) and external factors (user, department, etc.)
- May be preemptive or non-preemptive .
- SJF is an important case of priority scheduling, with priority inversely proportional to predicted next CPU burst length.
- May cause starvation, i.e. indefinite blocking of processes
- Aging: gradually increase the priority of a process waiting for a long time
- Priority inversion: a low-priority process gets the priority of a high-priority process waiting for it

Example:

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

Gantt chart:



AWT = 8.2 mS

Problem with priority scheduling algorithms is indefinite blocking or starvation. A solution to the problem of indefinite blockage of low priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example if priorities range from 0 (low) to 127 (high), we could increment the priority of a waiting process by 1 every 15 mins.

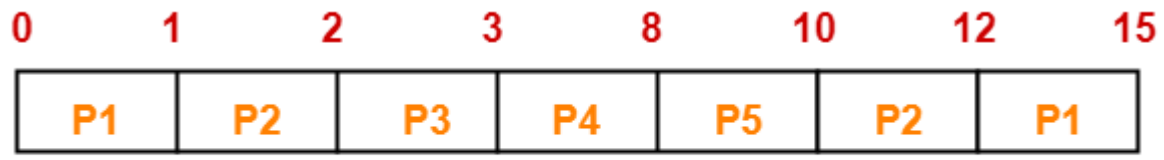
User Problems:

Problem 1:
Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turnaround time. (Higher number represents higher priority)
Solution-

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turnaround time – Burst time

Process Id	Exit time	Turn Around time	Waiting time

P1	15	15 – 0 = 15	15 – 4 = 11
P2	12	12 – 1 = 11	11 – 3 = 8
P3	3	3 – 2 = 1	1 – 1 = 0
P4	8	8 – 3 = 5	5 – 5 = 0
P5	10	10 – 4 = 6	6 – 2 = 4

Now,

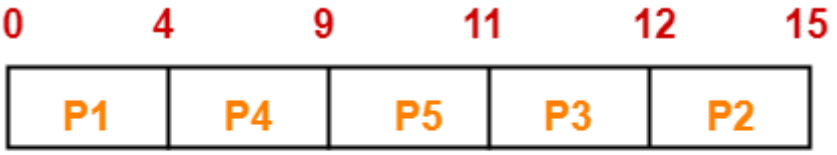
- Average Turnaround time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

Problem 2:
Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	**0**	**4**	**2**
P2	**1**	**3**	**3**
P3	**2**	**1**	**4**
P4	**3**	**5**	**5**
P5	**4**	**2**	**5**

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turnaround time. (Higher number represents higher priority)

Solution-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turnaround time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

Now,

- Average Turnaround time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Round-Robin Scheduling

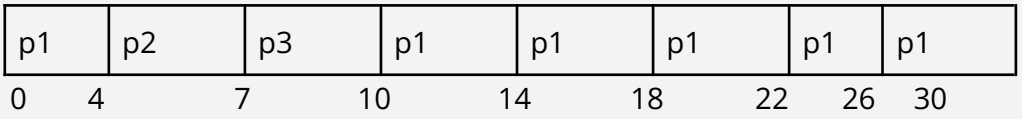
The round-robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice), is defined.
The ready queue is treated as a circular queue.

Example

Process	Burst Time
P1	24
P2	3
P3	3

Time quantum = 4 ms

Gantt chart



The average waiting time is 17/3 = 5.66 milliseconds.
Waiting time for P1 = 26 – 20 = 6
P2 = 4
P3 = 7 (6+4+7 / 3 = 5.66 ms)

The performance of the RR algorithm depends heavily on the size of the time–quantum. If the time-quantum is very large(infinite) then RR policy is the same as FCFS policy. If time quantum is very small, RR approach is called processor sharing and appears to the users as though each of n processes has its own processor running at 1/n the speed of a real processor.

User Problems:

Problem 1:

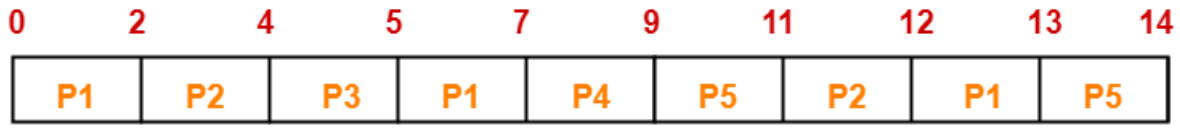
Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turnaround time.

Solution:

Ready Queue-
P5, P1, P2, P5, P4, P1, P3, P2, P1



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turnaround time – Burst time

Process Id	Exit time	Turn Around time	Waiting time

P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

Now,

- Average Turnaround time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

Problem 2:
Four jobs to be executed on a single processor system arrive at time 0 in the order A, B, C, D. Their burst CPU time requirements are 4, 1, 8, 1 time units respectively. The completion time of A under round robin scheduling with time slice of one time unit is-

1. **10**
2. **4**
3. **8**
4. **9**

Solution:

Process Id	Arrival time	Burst time
A	0	4
B	0	1
C	0	8
D	0	1

Ready Queue-

C, A, C, A, C, A, D, C, B, A

Gantt Chart

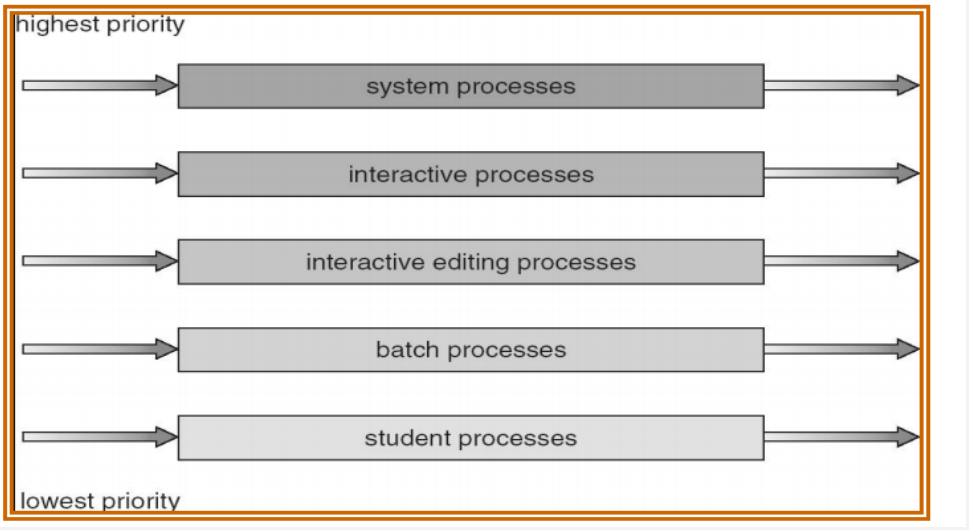
Clearly, completion time of process A = 9 unit.
Thus, Option (D) is correct.

Multilevel Queue Scheduling

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues . The processes are completely assigned to one queue, generally based on some unique property of the process, such as memory size, process priority, or process type.
There must be scheduling between the queues, which is commonly implemented as a fixed-priority preemptive scheduling.
For example the foreground queue may have absolute priority over the background queue.

Example : Five queens

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes



User Problems:

Problem 1:

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

Solution :

Process P needs 40 Seconds for total execution.

At Queue 1 it is executed for 2 seconds and then interrupted and shifted to queue 2.

At Queue 2 it is executed for 7 seconds and then interrupted and shifted to queue 3.

At Queue 3 it is executed for 12 seconds and then interrupted and shifted to queue 4.

At Queue 4 it is executed for 17 seconds and then interrupted and shifted to queue 5.

At Queue 5 it executes for 2 seconds and then it completes.

Hence the process is interrupted 4 times and completes on queue 5.

Multilevel Feedback Scheduling Notes

Multilevel Feedback Queue Scheduling

In the multilevel queue-scheduling algorithm, the processes are permanently assigned to the system entry queue. Processes do not move between queues. This setup has the advantage of lower scheduling overhead, but the lack of consistency.
Multilevel Feedback queue scheduling, however, allows the process to move between queues. The idea is to split processes with different CPU-burst features. If the process consumes too much CPU time, it will be moved to a much lower line. Similarly, the longest waiting process in the lowest line can be delivered to the most important line. This form of aging prevents starvation.

- Allows processes to move between queues
- Inter-queue scheduling: preemptive priority scheduling
- A process waiting too long in a low-priority queue may be moved to a high-priority queue.

Example:
consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process that arrives for queue 0 will, in turn, preempt a process in queue 1.

Printer Spooler Problem

As we know that a printer is a peripheral device, so it is slower in comparison to CPU and memory.So, if multiple users send some file to the printer to print then spooler comes into play.Spooler is a program in the printer which stores all the files coming to print and when printer is free it gives it to the printer in sequential manner.

Example:
This four line code is executed by each process in order to store its file in the spooler directory to print.

```
Load Ri, m[in]
store SD[Ri], "F-N"
INCR Ri
store m[in], Ri
```

in: Shared variable
m: Memory location
Ri: Register
F-N: File name
SD: Spooler directory

1. Line 1: In line one we are loading free memory location m[in], in register Ri
2. Line 2: In line two we are storing file name (F-N) in spooler directory (SD) at position Ri, which is for instance 0
3. Line 3: In line three we are incrementing the count of Ri from 0 to 1, so next file can be stored in at index 1
4. Line 4: In line four the new file will be stored at incremented memory location m[in]

So this was all about printer spooler problems. Hope you liked it and learned something new from it.

Real Life Example

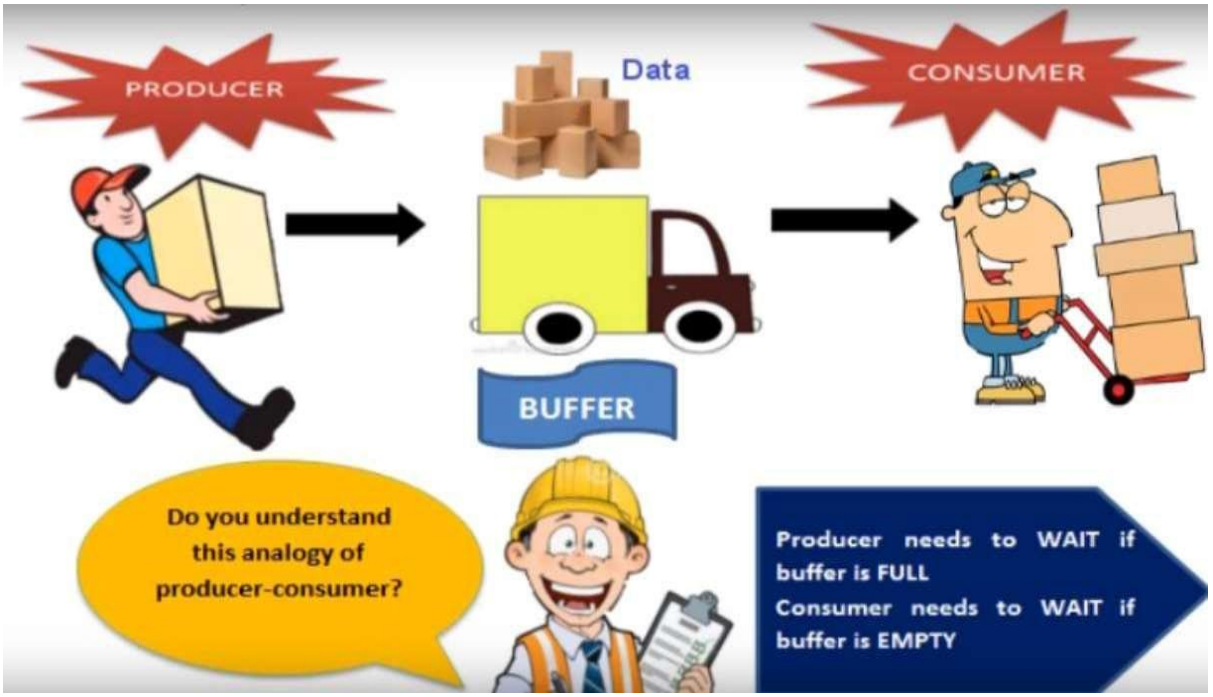
Managing and supporting a large enterprise's print infrastructure is a complex task. It is a huge challenge to manage printers in different buildings, different plants, even different countries. You don't know immediately when a printer is offline, jammed, out of toner or just swamped with too many print jobs. This lack of a single view of your enterprise print infrastructure leads to lost productivity at your company.

Problem In Operating System

Producer Consumer Problem

The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. In the problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently.
The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
Producer :- The producer's job is to generate a piece of data, put it into the buffer and start again.
Consumer :- The consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

Real Life Example:



Solution for the Producer - Consumer

- Producer wants to put but buffer- full Go to sleep and wake up when consumer takes one or more
- Consumer wants to take but buffer-empty go to sleep and wake up when producer puts one or more

The producer-consumer problem using semaphores

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

/*The most common real-life example of the Producer-Consumer algorithm is a process, called print spooling. Print spooling refers to putting jobs(in this case documents that are about to be printed)into a special location(buffer) in either computer memory, or hard disk, so that a printer could access this document whenever the printer is ready. There are a couple of advantages of spooling.
First of all the printer can access data from the buffer at any rate that is suitable for the printer.
Secondly the work of computer is not interrupted while printing, thus a user can perform other
tasks.

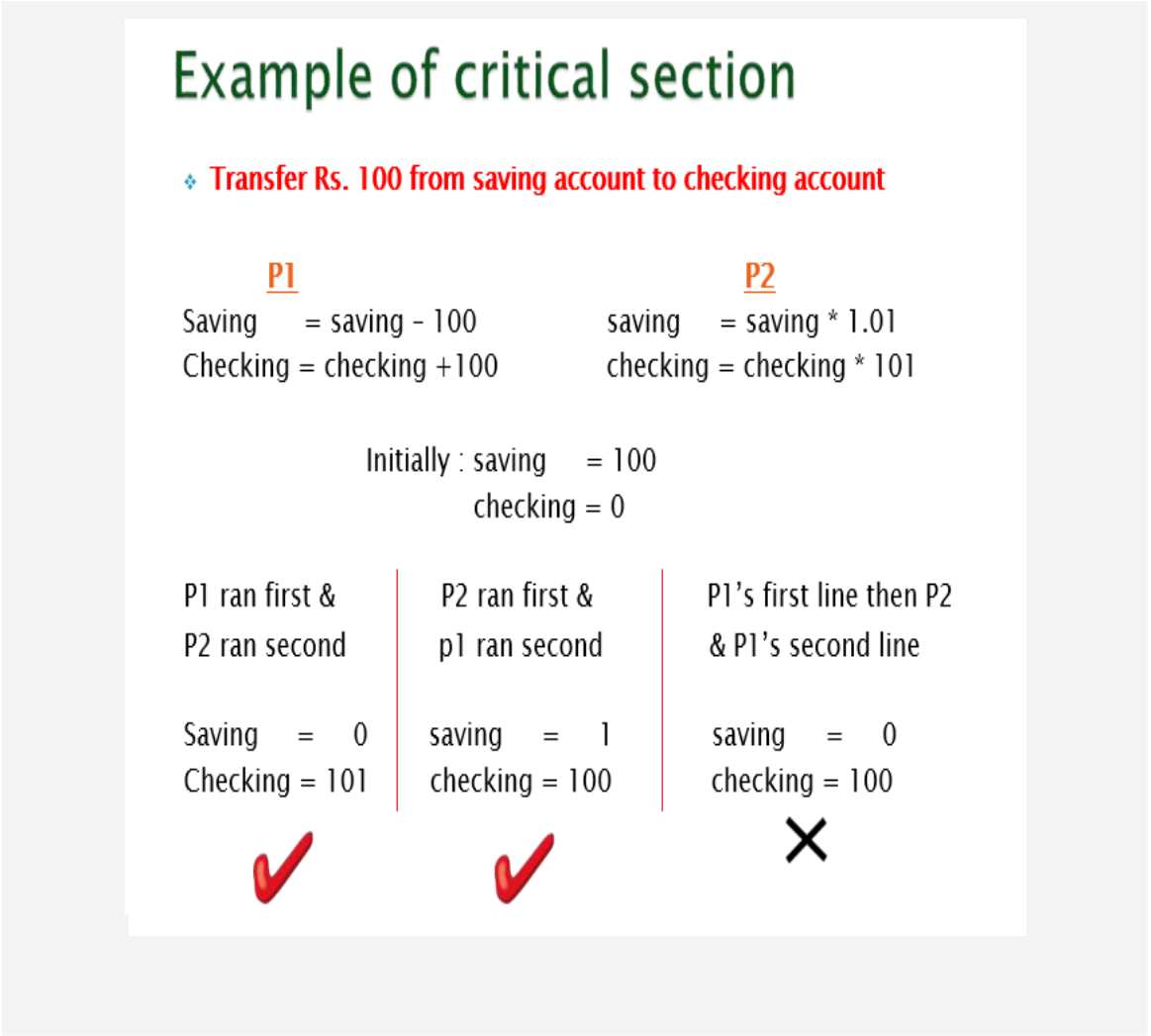
*/

Critical Section Problem

When a process is accessing shared modifiable data or a resource that can only operate on behalf of one process at a time , the process is said to be in a critical section. When one process is in a critical section , all other processes (at least those that access the shared modifiable data and/or resource) are excluded from their critical section.

The Critical Section Problem

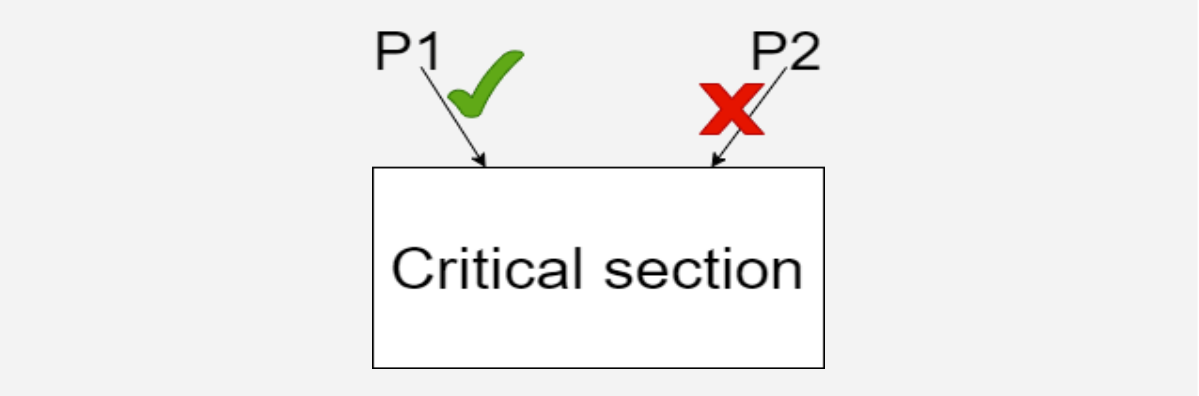
- n processes all competing to use some shared data
- Each process has a code segment, called the critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.



Solution To Critical Section Problem

1.Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

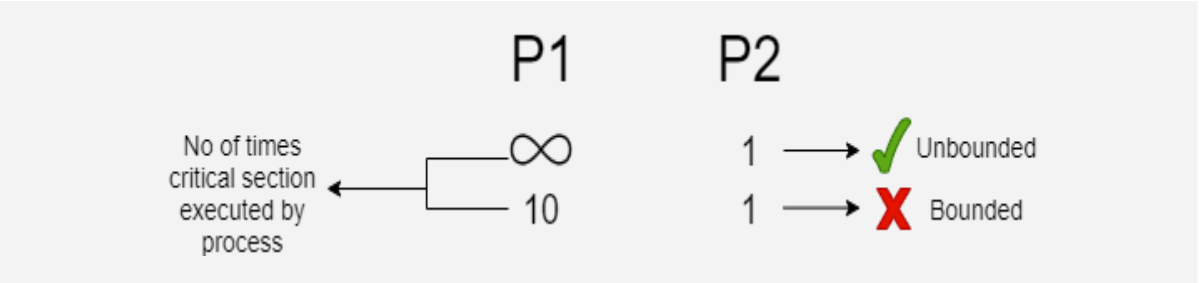
Suppose if P1 wants to execute a critical section and start executing it, and then at the same time (while P1 was executing critical section) P2 wants to enter the critical section in that case P2 will be blocked.
And by that we will achieve mutual exclusion.



2.Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
Example:
Suppose there is some common code written in the critical section and at present the critical section is empty (means that there is no process in it).
So, now P1 wants to enter the critical section and P2 is blocking it. Maybe because of some code written in the enter section of P2.
Therefore, in this case P2 is not going in the critical section and also blocking P1 to enter the critical section.
This means that progress is not there and this should not happen.

3.Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes.


P1
P2
No of times critical section executed by process
∞
10
1
Unbounded
1
Bounded

Semaphore

Semaphore

As it is difficult for the application programmer to use these hardware instructions, to overcome this difficulty we use the synchronization tool called Semaphore (that does not require busy waiting).Semaphore is a type of flexible or non-flexible data that is used to control access to common shared resources through multiple processes in the same system as a multitasking operating system.
It is a mechanism that can be used to provide synchronization of tasks. It is a low level synchronization mechanism.

A semaphore is an object that consists of a counter, a waiting list of processes, and two methods: signal and wait.All the modifications to the integer value of the semaphore in the wait() and signal() atomic operations must be executed indivisibly. i.e. when one process changes the semaphore value, no other process will change the same semaphore value simultaneously.

When the count for the semaphore is 0 means that all the resources are being used by some processes. Otherwise resources are available for the processes to allocate .
When a process is currently using a resource means that it blocks the resource until the count becomes > 0.
For example, let us consider two processes P1 and P2 and a semaphore initialized with a value of 1. The value of the semaphore becomes 0 if the process P1 enters the critical section. If the process P2 intends to enter the critical section then the value of the semaphore has to be greater than 0, until that the process has to wait. This is only possible when P1 completes the critical section and calls the Signal operation on the semaphore. Mutual exclusion is obtained this way.

Operation in Semaphore

1. **Wait**
2. **Signal**

Wait

This operation is used to control the entry of a task into the critical section.
The process that wishes to use a resource must perform the wait( ) operation (count is decremented) .

The definition for wait() is as follows:

```
wait (S)
{
        while S <= 0 ; // no-op
        S--;
}
```

Signal

This operation is used to control the exit of a task from a critical section. This increases the value of the argument by 1.
The process that releases a resource must perform the signal() operation (count is incremented) .

The definition for signal() is as follows:

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

Types of semaphore

1. **Counting Semaphore**
2. **Binary Semaphore**

Counting semaphore

The value of the Counting Semaphore can range over an unrestricted domain. Counting Semaphores are used to control the access of given resources each of which consists of some finite no. of instances. This counting semaphore is initialized to the number of resources available.

Binary semaphore

The value of the Binary Semaphore can range between 0 and 1 only.
In some systems the Binary Semaphore is called as Mutex locks, because they are locks to provide mutual exclusion. We can use the Binary Semaphore to deal with critical section problems for multiple processes.

Implementation of semaphore

The main disadvantage of the semaphore definition is, it requires the busy waiting. Because when one process is in the critical section and if another process needs to enter into the critical section must have to loop in the entry code continuously.

Implementation of semaphore with no busy waiting:

*****To overcome the need of the busy waiting we have to modify the definition of wait() and signal() operations. i.e. when a process executes wait() operation and finds that it is not positive then it must wait.
***** Instead of engaging the busy wait, the process blocks itself so that there will be a chance for the CPU to select another process for execution. It is done by block() operation. - Blocked processes are placed in the waiting queue.
*****Later the process that has already been blocked by itself is restarted by using wakeup() operation, so that the process will move from waiting state to ready state. - Blocked processes that are placed in the waiting queue are now placed into the ready queue.
*****To implement the semaphore with no busy waiting we need to define the semaphore of the wait() and signal() operation by using the 'C' Struct.
Which is as follows:

```
 typedef struct
{
int value;
struct process *list;
 }semaphore;
```

- i.e. each semaphore has an integer value stored in the variable "value" and the list of processes list.
- When a process performs the wait() operation on the semaphore then it will add a list of processes to the list .

- When a process performs the signal() operation on the semaphore then it removes the processes from the list.

Semaphore Implementation with no Busy waiting

Implementation of wait: (definition of wait with no busy waiting)

```
wait (S)
{
    Value--;
    if (value < 0)
   {
       add this process to waiting queue
       block();
   }
}
```

Implementation of signal: (definition of signal with no busy waiting)

```
Signal (S)
{
   Value++;
   if (value <= 0)
  {
          remove a process P from the waiting queue
           wakeup(P);
  }
}
```

Advantages

The different advantages of semaphores are given below:

- They do not allow more than one process to enter the critical section. In this way, mutual exclusion is achieved and thus they are extremely efficient than other techniques for synchronization.

- Due to busy waiting in semaphore, there is no wastage of process time and resources. This is because the processes are only allowed to enter the critical section after satisfying a certain condition.
- They are machine-independent as they run in the machine-independent code of the microkernel.
- They allow flexible management of resources.

Disadvantages

The different limitations of semaphores are given below:

- There may be a situation of priority inversion where the processes having low priority get access to the critical section than the processes having higher priority.
- To avoid deadlocks, the wait() and signal() operations have to be executed in the correct order.
- Semaphore programming is complicated and there are chances of not achieving mutual exclusion.

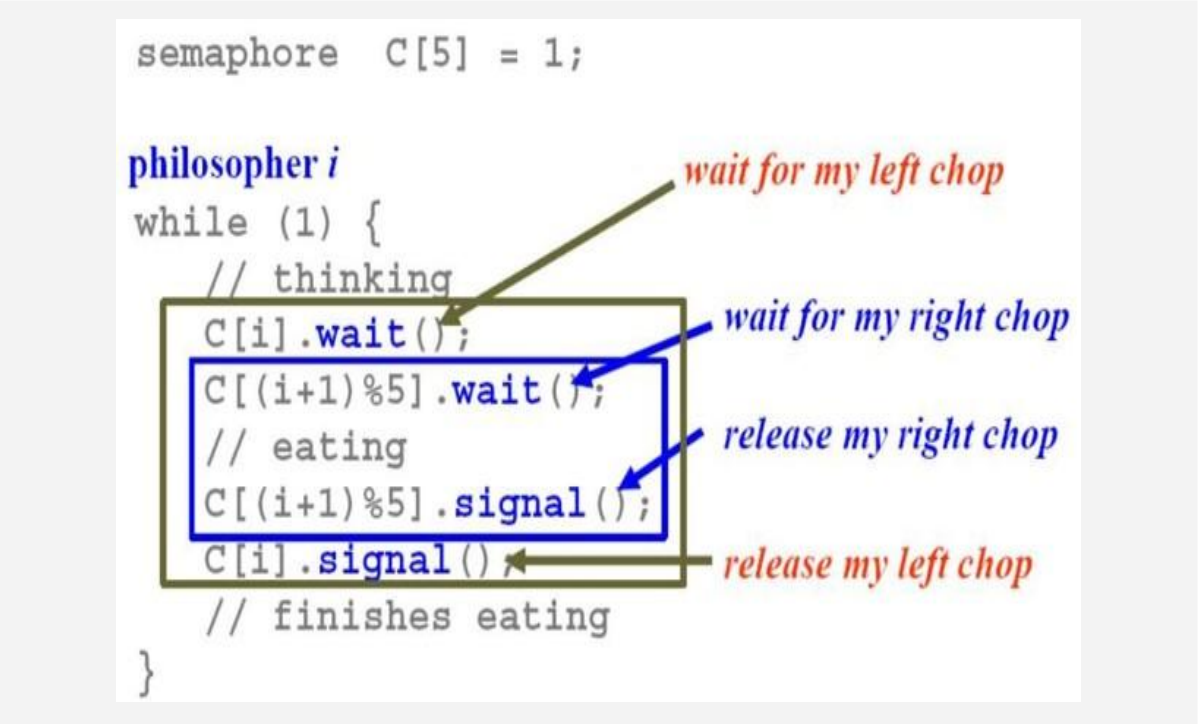
Problem in implementation of semaphore

Dining Philosophers Problem

The problem with dining philosophers is that there are five philosophers who share a round table and eat and think differently. There is a small bowl of rice for each philosopher and 5 chopsticks. The philosopher needs both their right and left chopsticks in order to eat. A hungry philosopher can only eat if there are both sticks available.Otherwise the philosopher puts down his stick and begins to think again. The dining philosopher is an old synchronization problem as it points to a large class of concurrency control problems.

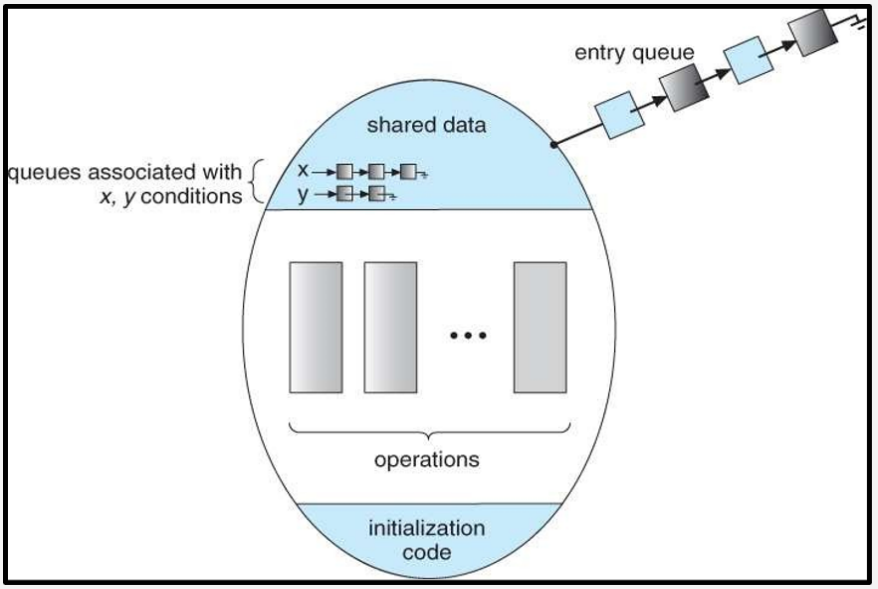
Solution using semaphore

- Chopsticks are shared items (by two philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1.
- A philosopher calls wait() before picking up a chopstick and calls signal() to release it.



Monitor

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met.



A monitor can have variables of the condition type that can be accessed by wait() and signal() operations only.
The operation x.wait() means that the process making this operation is suspended until another process invokes x.signal().
The operation x.signal() resumes exactly one suspended process and has no effect if there is none.

Syntax:

```
 monitor monitor_name
{
    // shared variable declarations initialization_code (...)
    { ... }
     procedure P1 (...)
     { ... }
     procedure Pn (...)
     { ... }
}
```

Peterson’s Solution

Peterson’s Solution

It is restricted to two processes that alternates the execution between their critical and remainder sections.
Consider that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
The two processes share two variables:
 * int turn;
 * Boolean flag[2]
The variable turn indicates whose turn it is to enter the critical section.
The flag array is used to indicate if a process is ready to enter the critical section.
flag[i] = true implies that process Pi is ready!
Note:- Peterson’s Solution is a software based solution.

Algorithm for Process Pi

```
while (true)
{
   flag[i] = TRUE;
   turn = j;
   while ( flag[j] && turn == j);
   CRITICAL SECTION flag[i] = FALSE;
   REMAINDER SECTION
 }

do
{
    acquire lock
            Critical Section
    release lock
            Remainder Section
```

```
}
while (True);
```

Note:-Race Conditions are prevented by protecting the critical region by the locks.

Two Process Working Concurrently

Process 1

```
do
{
  flag1 = TRUE;
  turn = 2;
  while (flag2 && turn == 2);
  critical section.....
  flag1 = FALSE;
  remainder section.....
 } while (1)
```

Process 2

```
do
{
  flag2 = TRUE;
  turn = 1;
  while (flag1 && turn == 1);
  critical section.....
  flag2 = FALSE;
  remainder section.....
} while (1)
```

Shared Variables }--- flag1, flag2, turn

Example

Process 0:
flag[0] := TRUE
turn := 1
check (flag[1] = TRUE and turn = 1)
*Condition is false because flag[1] = FALSE
* Since condition is false, no waiting in while loop
* Enters the critical section

Phase-1

Process 1:
flag[1] := TRUE
turn := 0
check (flag[0] = TRUE and turn = 0)
*Since condition is true, it keeps busy waiting until it loses the processor
*Process 0 resumes and continues until it finishes in the critical section

Phase-2

Process 0: *Leaves critical section Sets flag[0] := FALSE * Start executing the remainder (anything else a process does besides using the critical section) * Process 0 happens to lose the processor **Phase-3**	**Process 1:** check (flag[0] = TRUE and turn = 0) * This condition fails because flag[0] = FALSE * No more busy waiting *Enter the critical section **Phase-4**

Implementation

```
public class cSection {
int turn;
boolean flag[] = new boolean[2];  int i = 0, j = 1;
// CSC variables
int counter = 0;// counter for giving processes an upper bound
int cscVar = 13;

private class ProcessI extends Thread { // process thread for i
@Override
public void run() {
try {
do {
flag[i] = true;  turn = j;
while (flag[j] && turn == j)
; // wait for j to finish
// critical section
System.out.println("I is in critical section");  cscVar++;
System.out.println(cscVar);  counter++;
System.out.println("counter is " + counter + "n      ");
//
flag[i] = false;
// remainder section
```

```
} while (counter < 100); // 100 is upper bound, to remove
// infinite looping
}
catch (Exception ex) {  ex.printStackTrace();
}
}
}

private class ProcessJ extends Thread { // process thread for j
@Override
public void run() {
try {
do {
flag[j] = true;  turn = i;
while (flag[i] && turn == i)
;// wait for i to finish
// critical section
System.out.println("J is in critical section");  cscVar--;
System.out.println(cscVar);  counter++;
System.out.println("counter is " + counter + "n      ");
//
flag[j] = false;
// remainder section
} while (counter < 100); // 100 is upper bound, to remove
// infinite looping
}
catch (Exception ex) {  ex.printStackTrace();
}
}
}
public cSection() {
System.out.println("Starting Threads/Processes");  Thread I = new ProcessI();
Thread J = new ProcessJ();  I.start(); // start process i  J.start(); // start process j
}
public static void main(String[] args) {  cSection cSec = new cSection();
}
}
```

Test and Set Solution Notes

Test and Set Solution for Critical Section Problem

In this, shared key variables are initialized to false. The TestAndSet (lock) algorithm works this way - it always returns any value sent to it and sets the lock to reality. The first process will go into a critical phase simultaneously as TestAndSet (lock) will return false and exit the time loop. Some operations cannot enter now as the lock is set to true so while the loop remains true. When the first process comes out of a critical phase, the lock is converted into a false one. So, now some processes can go in one step at a time. There is progress. However, after the initial procedure, any procedure can be entered. No queue has been saved, so any new system that detects whether a key is false again may be incurred. The restricted wait is not guaranteed.

TestAndSet Instruction

Definition:
boolean TestAndSet (boolean *target) @{ @ boolean rv = *target;@ *target = TRUE;@ return rv:@ }@ Solution using TestAndSet Shared boolean variable lock., initialized to false. @Solution:@ do @{@ while (TestAndSet (&lock)) ; // do nothing // critical section @lock = FALSE; // remainder section @} while (TRUE);

Implementation

This algorithm solves the critical section problem for n processes using a Test and Set instruction (called TaS here). This instruction does the following function atomically:

```
function TaS(var Lock: boolean):
boolean;
    begin
        TaS := Lock;
        Lock := true;
    end;
```

The solution is:

```
1 var waiting: shared array [0..n-1] of
boolean;
 2    Lock: shared boolean;
 3    j: 0..n-1;
 4    key: boolean;
       ...
 5 repeat     (* process Pi *)
 6    waiting[i] := true;
 7    key := true;
 8    while waiting[i] and key do
 9       key := TaS(Lock);
10    waiting[i] := false;
11    (* critical section goes here *)
```

```
12     j := i + 1 mod n;
13     while (j <> i) and not waiting[j]
do
14         j := j + 1 mod n;
15     if j = i then
16         Lock := false
17     else
18         waiting[j] := false;
19 until false;
```

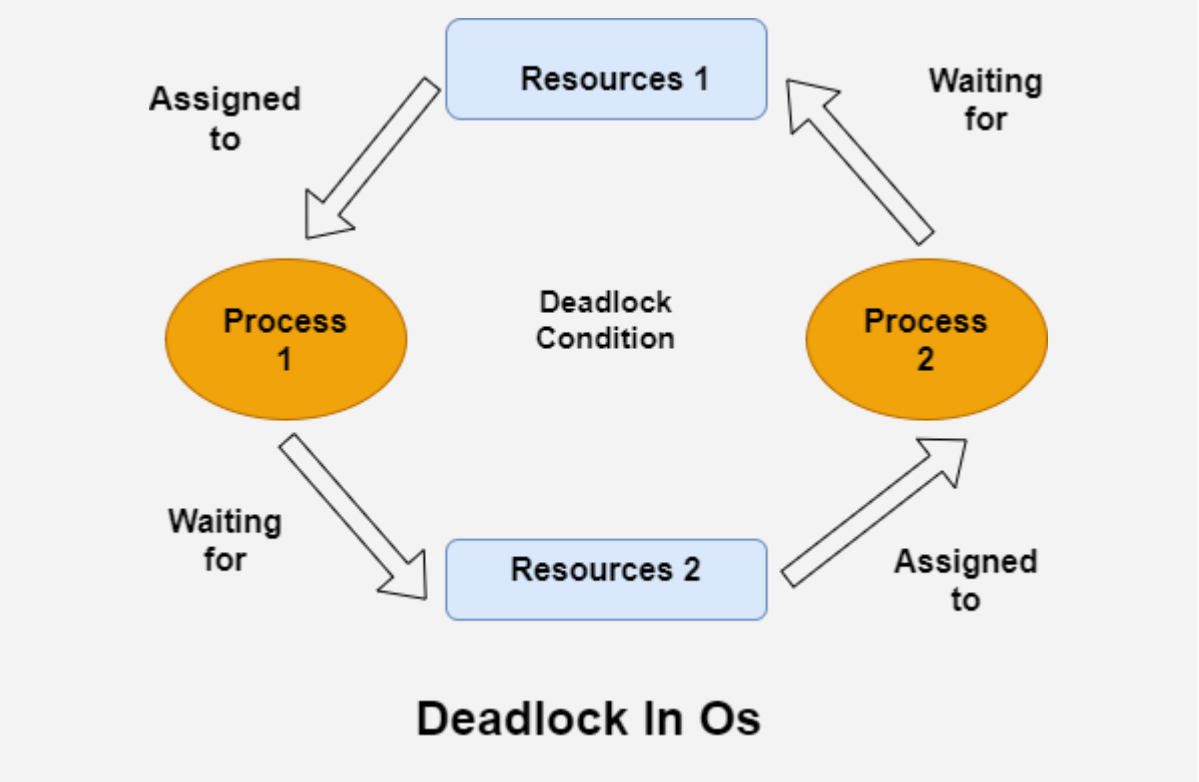
lines 1-2: These are global to all processes, and are all initialized to false.
lines 3-4: These are local to each process i and are uninitialized.
lines 5-10: This is the entry section. Basically, waiting[i] is true as long as process i is trying to get into its critical section; if any other process is in that section, then Lock will also be true, and process i will loop in lines 8-9. Once the process i can go on, it is no longer waiting for permission to enter, and sets waiting[i] to false (line 10); it then proceeds into the critical section. Note that Lock is set to true by the TaS instruction in line 9 that returns false.
lines 12-18: This is the exit section. When process i leaves the critical section, it must choose which other waiting process may enter next. It starts with the process with the next higher index (line 12). It checks each process to see if that process is waiting for access (lines 13-14); if no-one is, it simply releases the lock (by setting Lock to false; **lines 15-16).** However, if some other process process j is waiting for entry, process i simply changes waiting[j] to false to allow process j to enter the critical section (lines 17-18).

Deadlock

Deadlock

In a multiprogramming system, several processes may compete for a finite number of resources. A process requests for resources, and if the resources are not available at the time then the process enters the waiting state. Sometimes, a process will wait indefinitely because the resources it has requested for are being held by other similar waiting processes.
Deadlock is a state in which two or more processes are waiting indefinitely because the resources they have requested for are being held by one another.

A process is deadlocked if it is waiting for an event which is never going to happen. Deadlocks can occur via system calls, locking, etc.



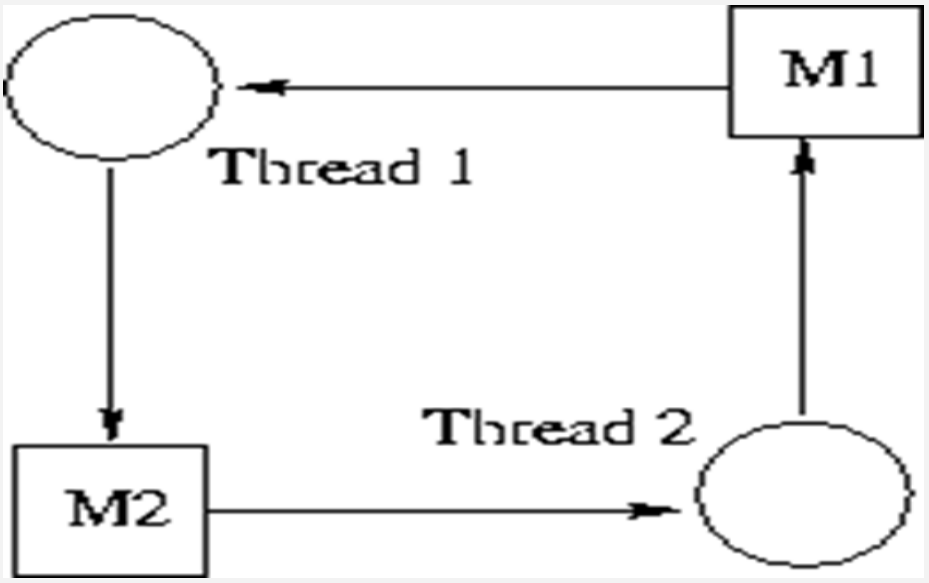
Example of deadlock

```
Let S and Q be two semaphores initialized to 1
     P0                  P1
wait (S);              wait (Q);
wait (Q):              wait (S);
.
.
.
signal (S);            signal (Q);
signal (Q);            signal (S);
```

Conditions for Deadlock

Mutual Exclusion

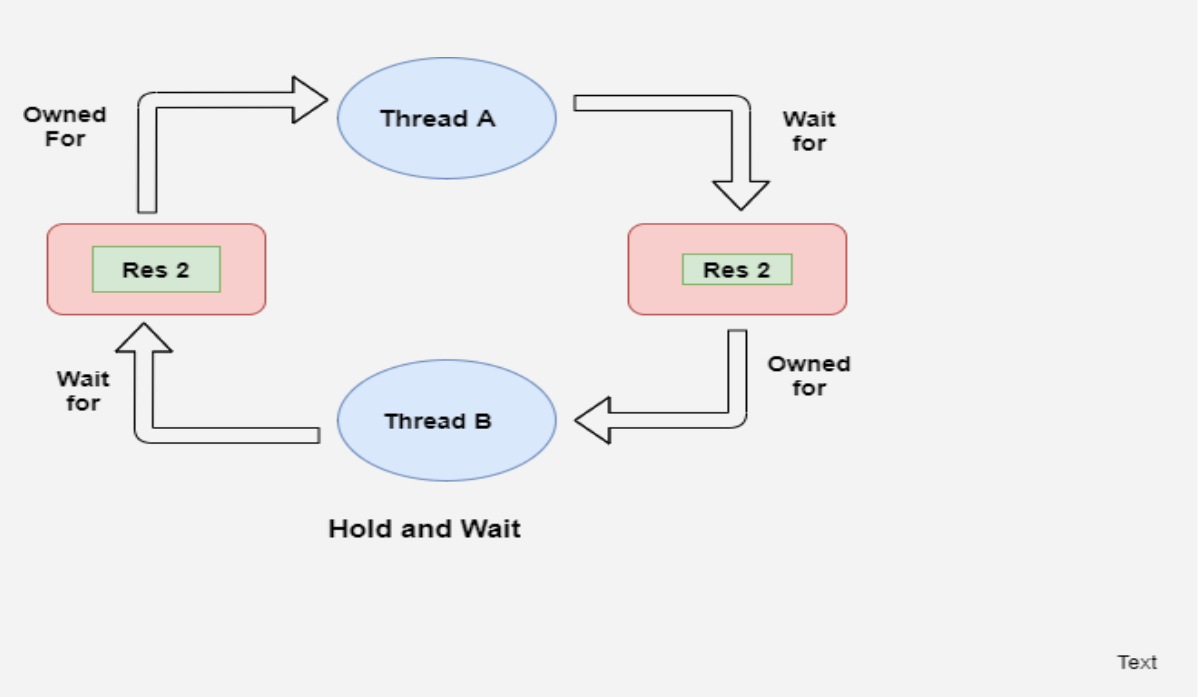
- The resources involved must be unshareable.
- Every resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource).



Mutual Exclusion

Hold and Wait

There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

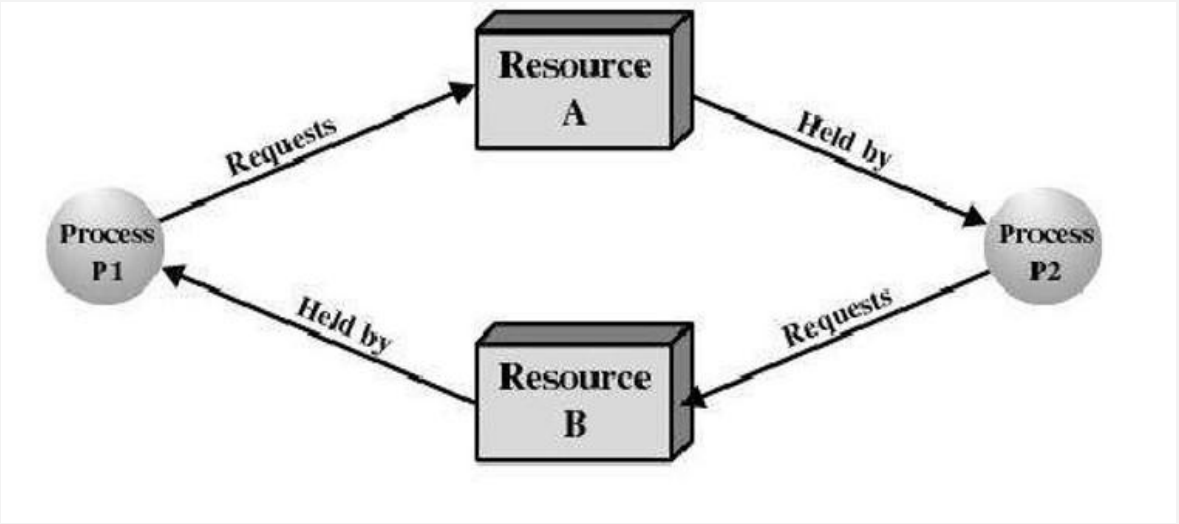


No Preemption Condition

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released implicitly. Then the preempted resources are added to the list of resources for which the process is waiting.

Circular Wait

In circular wait a chain of processes exists in which each process waits for one or more resources held by the next process in the chain.



Circular Wait

Methods Of Deadlock Handling

Banker's Algorithm

This algorithm can be used in a bank to ensure that the bank never allocates the available money in a way that it could no longer satisfy the needs of all its clients. A new task must declare the maximum number of instances of each resource type that it may need. This number should not exceed the total number of instances of that resource type in the system.
When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state. If yes, then the resources may be allocated to the process. If not, then the process must wait till other processes release enough resources.

Data Structures for the Banker's Algorithm

Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available
Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj
Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task Need [i,j] = Max[i,j] – Allocation [i,j]
Finish: Boolean value, either true or false. If finish[i]=true for all i return safe else unsafe

Safety Algorithm
1.Let Available and Finish be vectors of length m and n, respectively.
Initialize: Finish [i] = false for i = 0, 1, ..., n- 1

2.Find an i such that both: (a) Finish [i] = false (b) Needi Available If no such i exists, go to step 4
3. Available = Available + Allocation Finish[i] = true go to step 2
4.If Finish [i] == true for all i, then the system is in a safe state

Resource-request algorithm checks if a request can be safely granted. Pi is requesting for more resources and Request[m] be the request.

1. If Request > Need[i], then error
2. If Request > Available, then wait
3. Pretend to allocate the request

    Available = Available - Request

    Allocation[i] = Allocation[i] + Request

    Need[i] = Need[i] - Request

If the resultant state is safe then the resources are actually allocated, else values of Available, Allocation[i] and Need[i] are restored to their previous values.
Time complexity = O(m).

Ostrich Algorithm

- The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all.
- Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs.
- Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is.
- If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.
- Most operating systems, including UNIX and Windows, just ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything.
- If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

Resource preEmption

To eliminate deadlocks using resource preemption, preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
There are 3 methods to eliminate the deadlocks using resource preemption. These are :
a) SELECTING A VICTIM : Select a victim resource from the deadlock state, and preempt that one.
b) ROLLBACK : If a resource from a process is preempted, what should be done with that process. The process must be roll backed to some safe state and restart it from that state.
c) STARVATION : It must be guaranteed that resources will not always be preempted from the same process to avoid starvation problem.

Memory Management

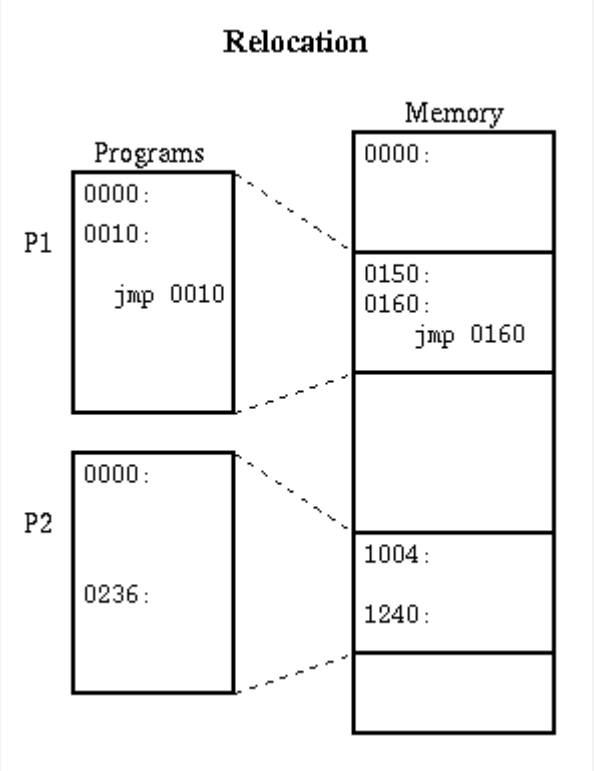
Logical and Physical Address Space

Logical address is the one that is generated byLogical address is the one that is generated by CPU, also referred to as virtual address. TheCPU, also referred to as virtual address. The program perceives this address space.program perceives this address space.Logical address space is the set of all logical addresses generated by a program.

Physical address is the actual addressPhysical address is the actual address understood by computer hardware i.e., memory understood by computer hardware i.e., memory unit. Logical to physical address translation unit.Physical address space is the set of all physical addresses generated by a program.

Program Relocation

The term program relocatability refers to the ability to load and execute a given program into memory.Relocation is a way to map virtual addresses into physical addresses.
When a task is loaded in main memory since there are several instructions inside the process so here addresses of these different instructions inside the process are relocatable addresses which are converted into actual addresses by the loader in software approach.
But there is some problem with this approach: suppose if a process is first loaded then removed and then after loaded again so in this situation the loader will get confused.
So to avoid this problem the operating system uses another method for relocation.
Instead of using this load time binding operating system used runtime binding.



Static Relocation

At load time, the OS adjusts the addresses in a process to reflect its position in memory. This method is the slow process because it involves software translation. It is used only once before the initial loading of the program.

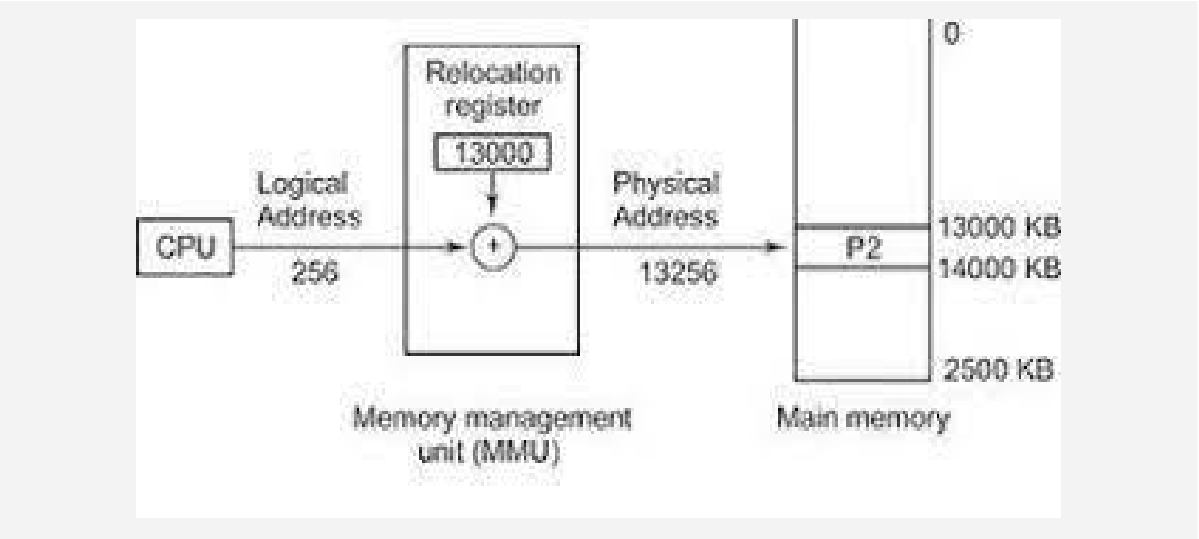
Once a process is assigned a place in memory and starts executing it, the OS cannot move it.

Dynamic Relocation:

hardware adds relocation register (base) to virtual address to get a physical address.
hardware compares address with limit register (address must be less than base).
If test fails, the processor takes an address trap and ignores the physical address

Relocation Register

Relocation register is a special register in the CPU used for program relocation means mapping of logical addresses used by the program to physical addresses of the system's main memory.



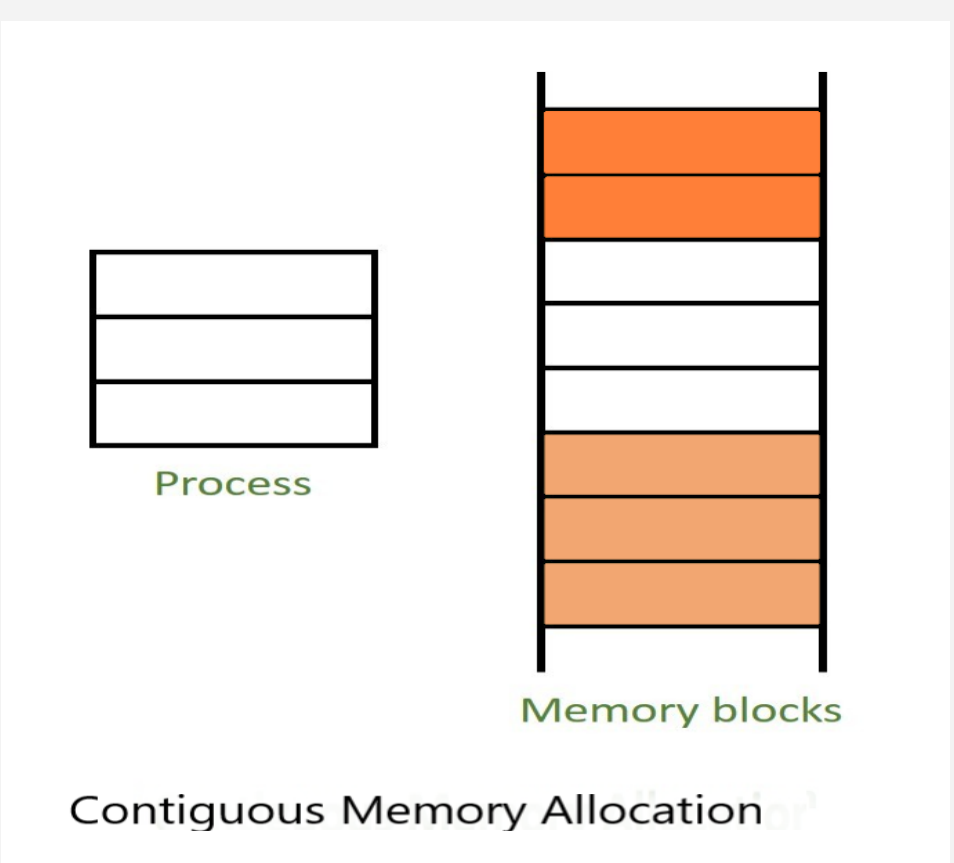
For example, if the base(relocation register content) is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000 and an access to location 346 is mapped to location 14346.

Storage Allocation Methods in Memory Management

Storage Allocation and Management TechniquesThe Storage allocation can be of two types:
(i) Contiguous storage allocation.
(ii) Non-contiguous storage allocation.

Contiguous Storage Allocation

► Contiguous storage allocation implies that a program's data and instructions are assured to occupy a single contiguous memory area.
► It is further subdivided into Fixed-partition storage allocation strategy and variable-partition storage allocation strategy.



Contiguous:Divide the memory into fixed-sized blocks

1. Fixed-partition contiguous storage allocation

The processes with small address space use small partitions and processes with large address space use large partitions. This is known as fixed partition contiguous storage allocation.

2. Variable - partition contiguous storage allocation

This notion is derived from parking of vehicles on the sides of streets where the one who manages to enter will get the space. Two vehicles can leave a space between them that cannot be used by any vehicle. This means that whenever a process needs memory, a search for the space needed by it, is done. If contiguous space is available to accommodate that process, then the process is loaded into memory.

External Fragmentation

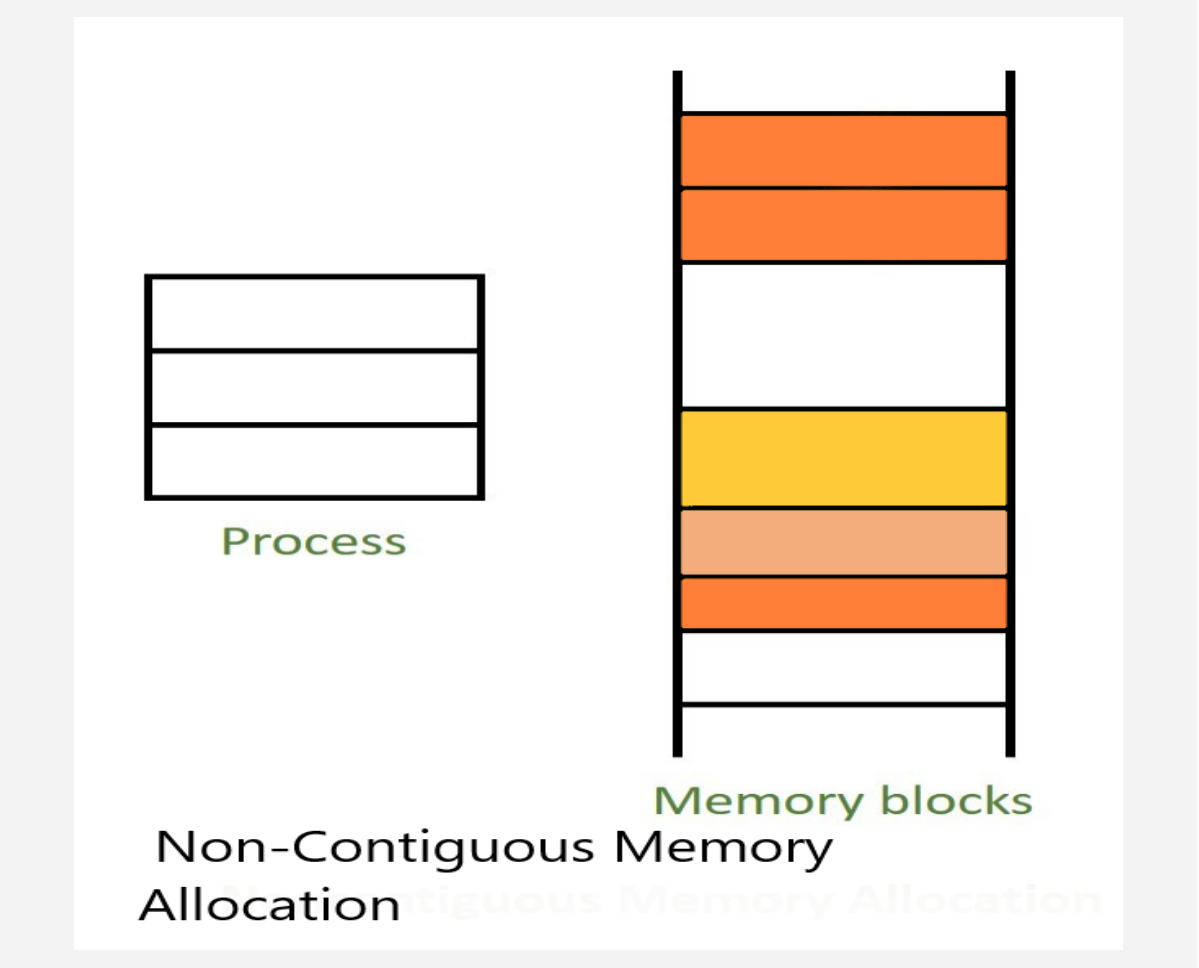
This phenomenon of entering and leaving the memory can cause the formation of unusable memory holes (like the unused space between two vehicles). This is known as External Fragmentation.

Non-contiguous Storage Allocation

To resolve the problem of external fragmentation and to enhance the degree of multiprogramming to a greater extent, it was decided to sacrifice the simplicity of allocating contiguous memory to every process. It was decided to have a non-contiguous physical address space of a process so that a process could be allocated memory wherever it was available.

There are 2 techniques for non-contiguous allocation:

1. Paging
2. Segmentation



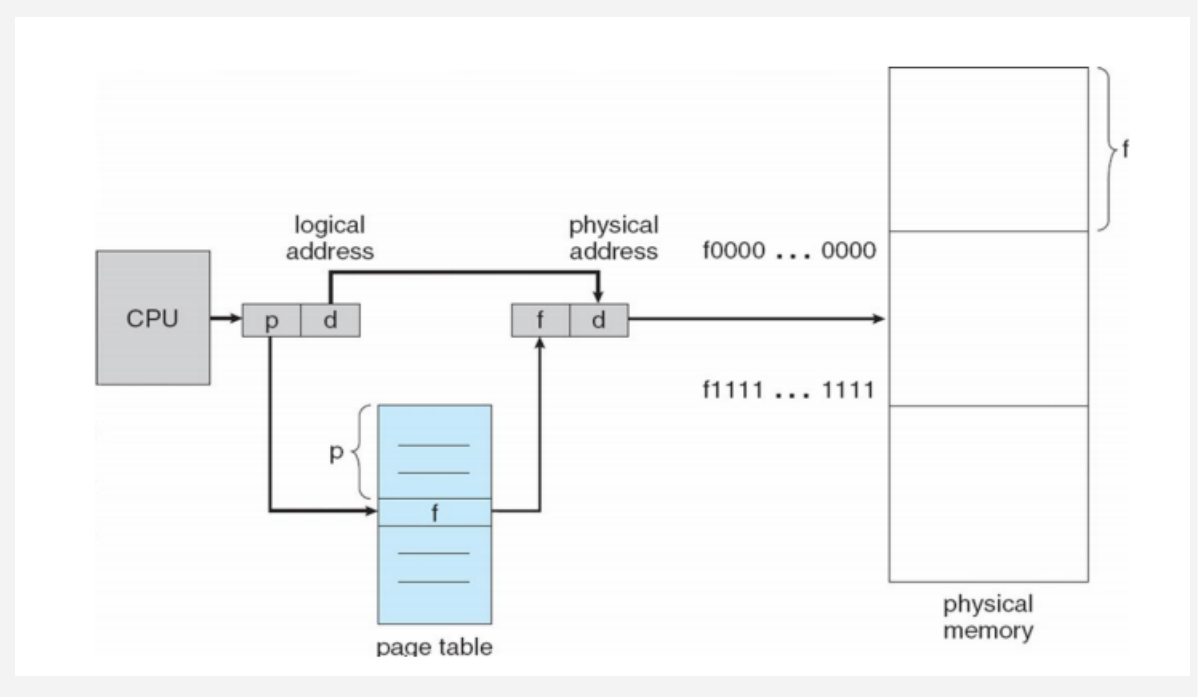
Paging

In paging, physical memory is divided into fixed-size blocks called frames and the logical memory is divided into the fixed-sized blocks called pages.

The size of a page is the same as that of a frame. The key idea of this method is to place the pages of a process into the available frames of memory, whenever, this process is to be executed. The address mapping is done by Page table.

Physical memory is divided into fixed size- blocks called FRAMES. (size is power of 2, for example 512 bytes)

Logical memory is divided into blocks of the same size called PAGES.



Example

Suppose, if the main memory size is 16 KB and Frame size is 1 KB. Here, the main memory will be divided into the collection of 16 frames of 1 KB each.

Advantages and Disadvantages of paging

Advantages :

no external fragmentation.
Simply memory management algorithm.
Swapping is easy(equal sized pages and page frame).

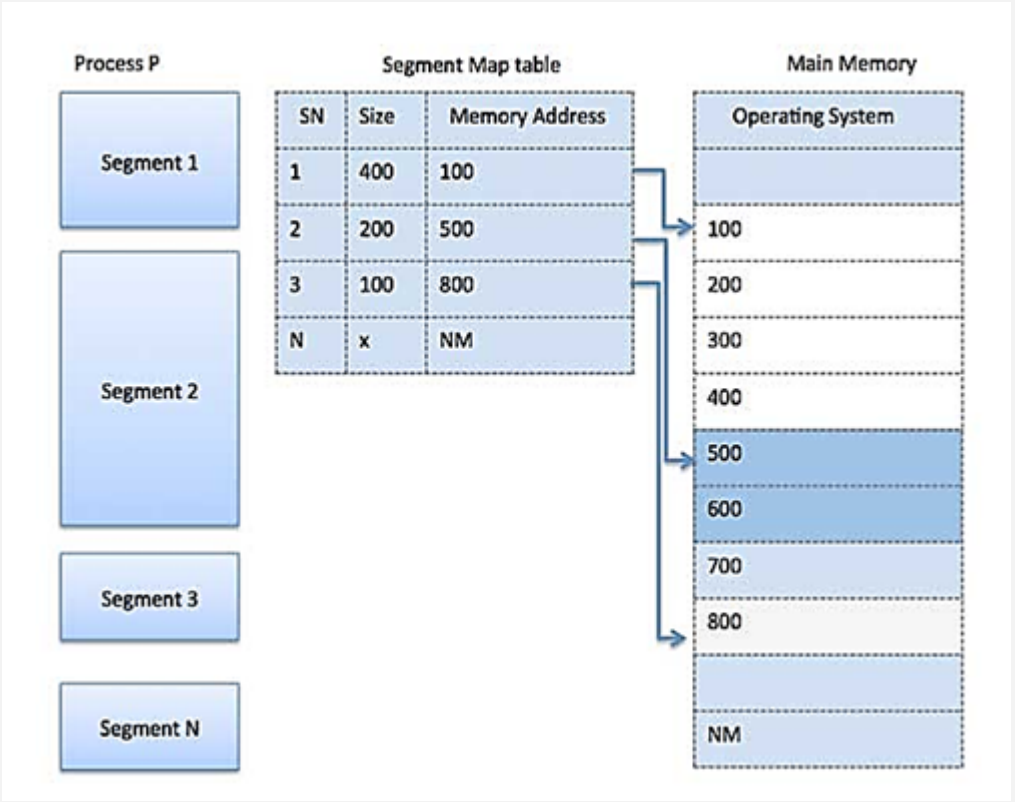
Disadvantages:

Internal fragmentation.
Page tables may consume more memory.

Segmentation

segmentation is another technique for the noncontiguous storage allocation. It is different from paging as it supports users' view of his program.

For a programmer it might be more relevant to divide the logical address space of his program into variable sized segments (with respect to his view of main program, subroutines, data, etc.) than to divide it into fixed size pages. Such variable sized segments, which are a collection of logically related information, are the basis of segmentation technique.



segmentation

For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.

Advantages and disadvantages of segmentation

Advantages :

no internal fragmentation.

Segment tables consume less memory than pages.
Lends itself to sharing data among processes.
Lends itself to protection.

Disadvantages: Costly memory management algorithm.

Paging

Paging

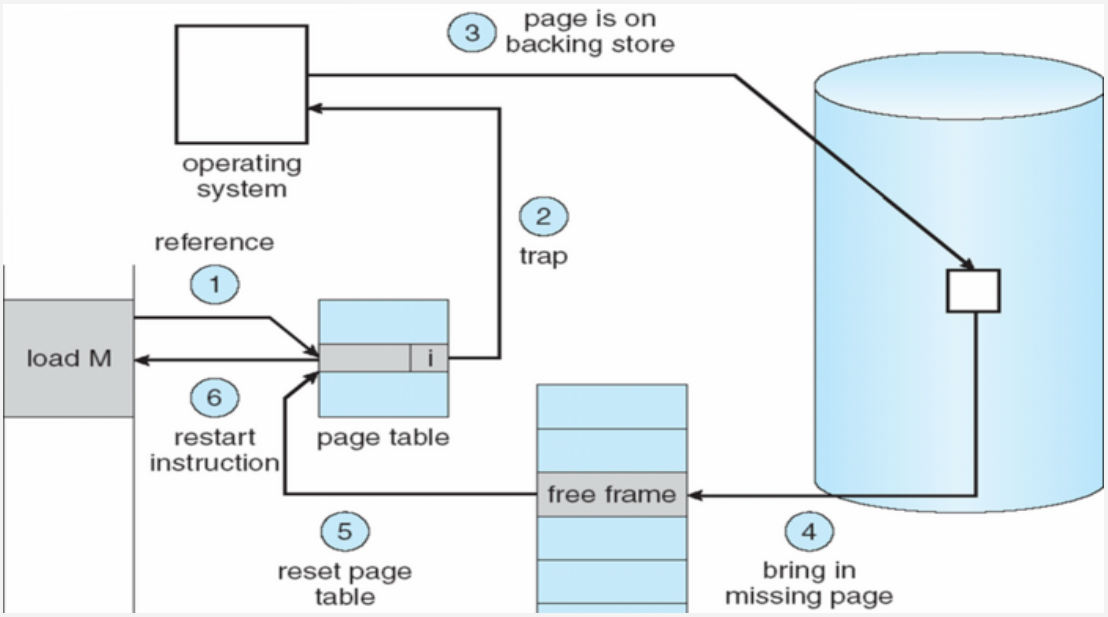
Memory Management technique that permits the physical address space of a process to be non contiguous is known as paging.Paging is used for faster access to data, and it is a logical concept.

Virtual memory

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Page faults

Page fault dominates like an error. If any program tries to access a piece of memory but which does not exist in physical memory, meaning main memory, then page fault will occur. The fault specifies the O/S that it must trace the all data into virtual memory management, and after that moves it from secondary memory like a hard disk to primary memory of the system.



Page fault

Handling of a Page Fault

1. Check the location of the referenced page in the PMT
2. If a page fault occurred, call on the operating system to fix it
3. Using the frame replacement algorithm, find the frame location
4. Read the data from disk to memory
5. Update the page map table for the process
6. The instruction that caused the page fault is restarted when the process resumes execution.

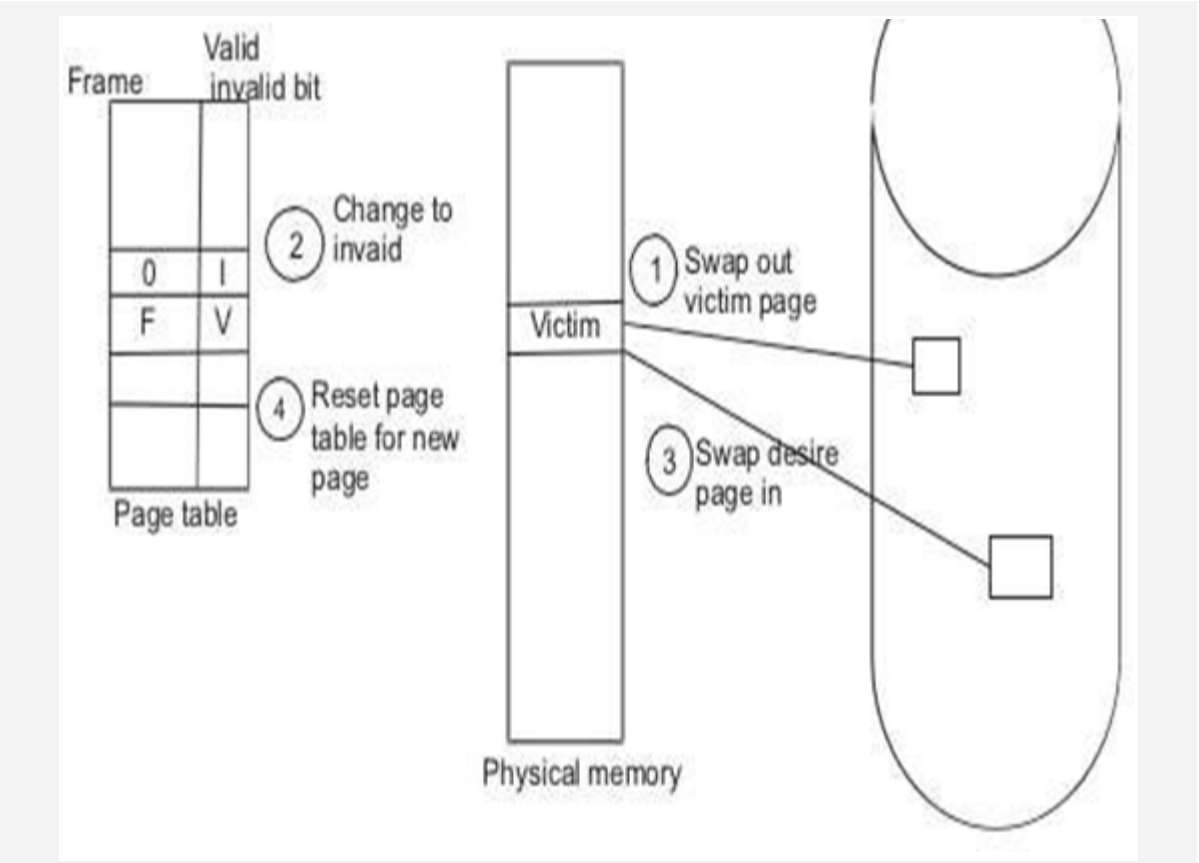
Page Replacement Algorithm

What is page Replacement?

When memory located in secondary memory is needed, it can be retrieved back to main memory.

Process of storing data from main memory to secondary memory ->swapping out

Retrieving data back to main memory ->swapping in



Page Replacement

Why do we need a page replacement algorithm?

The main goal of page replacement algorithms is to provide lowest page fault rate

Algorithms

- First In First Out
- Optimal Replacement
- Not Recently Used
- Second Chance
- CLOCK
- Not Frequently Used
- Least Recently Used
- Random Replacement

First-In First-Out (FIFO)

- Pages in main memory are kept in a list.
- Newest page is in head and the oldest in tail.
- It does not take advantage of page access patterns or frequency.

Optimal Replacement (OPT)

- When the memory is full, evict a page that will be unreferenced for the longest time.
- The OS keeps track of all pages referenced by the program.
- Only if the program's memory reference pattern is relatively consistent.

Not Recently Used (NRU)

- It favours keeping pages in memory that have been recently used.
- The OS divides the pages into four classes based on usage during the last clock tick:
    3. Referenced, modified
    2. Referenced, not modified
    1. Not referenced, modified
    0. Not referenced, not modified
- Pick a random page from the lowest category for removal,i.e. the not referenced, not modified page

Second Chance

- Modified version of FIFO
- Instead of swapping out the last page, the referenced bit is checked
- Gives every page a "second-chance"

Clock

- Modified version of FIFO
- The set of frame candidates for replacement is considered as a circular buffer.

Least Recently Used (LRU)

- It swaps the pages that have been used the least over a period of time.
- It is free from Belady’s anomaly

Not frequently used (NFU)

- This page replacement algorithm requires a counter
- The counters keep track of how frequently a page has been used
- The page with the lowest counter can be swapped out

Random

- This algorithm replaces a random page in memory.
- It fares better than FIFO.

Disk Scheduling Algorithm

First Come First Serve (FCFS)

It is the simplest form of disk scheduling algorithms. The I/O requests are served or processed according to their arrival. The request arrives first and will be accessed and served first.

Example: Given the following track requests in the disk queue, compute for the Total Head Movement2 (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, 64 .

Consider that the read/write head is positioned at location 50. Prior to this track location 199 was serviced. Show the total head movement for a 200 track disk (0-199).
Solution:
Total Head Movement Computation:
(THM) = (180 - 50) + (180-34) + (119-34) + (119-11) + (123-11) + (123-62) + (64-62)
= 130 + 146 + 85 + 108 + 112 + 61 + 2
(THM) = 644
tracks Assuming a seek rate of 5 milliseconds is given, we compute for the seek time using the formula:
Seek Time = THM * Seek rate
= 644 * 5 ms
Seek Time = 3,220 ms

There are some requests that are far from the current location of the R/W head which causes the access arm to travel from innermost to the outermost tracks of the disk or vice versa. In this example, it had a total of 644 tracks and a seek time of 3,220 milliseconds. Based on the result, this algorithm produced a higher seek rate since it follows the arrival of the track requests.

Shortest Seek Time First (SSTF)

This algorithm is based on the idea that the R/W head should proceed to the track that is closest to its current position. The process would continue until all the track requests are taken care of.

Example: Given the following track requests in the disk queue, compute for the Total Head Movement2 (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, 64

Consider that the read/write head is positioned at location 50. Prior to this track location 199 was serviced. Show the total head movement for a 200 track disk (0-199).

Solution:
(THM) = 12+2+30+23+84+24+4+57
(THM) = 236 tracks

Seek Time = THM * Seek rate
= 236 * 5ms
Seek Time = 1,180 ms

In this algorithm, request is serviced according to the next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue up to the last track request. There are a total of 236 tracks and a seek time of 1,180 ms, which seems to be a better service compared with FCFS .

SCAN

This algorithm is performed by moving the R/W head back-and-forth to the innermost and outermost track. As it scans the tracks from end to end .

Example: Given the following track requests in the disk queue, compute for the Total Head Movement2 (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, 64

Solution:
(THM) = (50-0) + (180-0) = 50 + 180

(THM) = 230
Seek Time = THM * Seek rate
= 230 * 5ms
Seek Time = 1,150 ms
This algorithm works like an elevator does.

In the algorithm example, it scans down towards the nearest end and when it reaches the bottom it scans up servicing the requests that it did not get going down. If a request comes in after it has been scanned, it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks and a seek time of 1,150. This is optimal than the previous algorithm.

Look

The disk arm starts at the first I/O request on the disk, and moves toward the last I/O request on the other end, servicing requests until it gets to the other extreme I/O request on the disk, where the head movement is reversed and servicing continues. It moves in both directions until both last I/O requests; more inclined to serve the middle cylinder requests.

Example: Given the following track requests in the disk queue, compute for the Total Head Movement2 (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, 64 ALI
Solution:
(THM) = (50-11) + (180-11) = 39 + 169
(THM) = 208 tracks

Seek Time = THM * Seek rate
= 208 * 5ms
Seek Time = 1,040 ms

This algorithm has a result of 208 tracks and a seek rate of 1,040 milliseconds. This algorithm is better than the previous algorithm.

C-Scan

The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.
Provides a more uniform wait time than SCAN; it treats all cylinders in the same manner.

C-Look

Look version of C-Scan.
Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.
In general, Circular versions are more fair but pay with a larger total seek time.
Scan versions have a larger total seek time than the corresponding Look versions.

RSS

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

LIFO

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

- Maximizes locality and resource utilization

Disadvantages

- Can seem a little unfair to other requests and if new requests keep coming in, it causes starvation to the old and existing ones.

N-STEP SCAN

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

Advantages

It eliminates starvation of requests completely

FSCAN

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

Advantages

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