# GROUP – A
## EXPERIMENT NO.: 01

**Title:**

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives.

**Objectives :**

- To understand Data structure of Pass-1 assembler
- To understand Pass-1 assembler concept
- To understand Advanced Assembler Directives

**Problem Statement :**

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature.

**Software Requirements:**

Latest jdk., Eclipse

**Theory**

**Concepts:**

**Introduction:**

There are two main classes of programming languages: high level (e.g., C, Pascal) and low level. Assembly Language is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

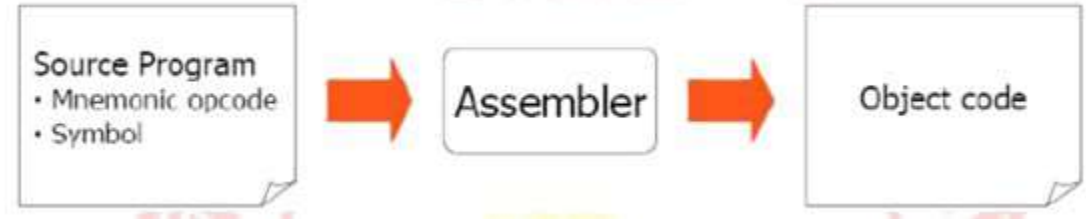



**Figure 1**. Executable program generation from an assembly source code

**Advantages of coding in assembly language are:**

- Provides more control over handling particular hardware components
- May generate smaller, more compact executable modules
- Often results in faster execution

**Disadvantages:**

1. Not portable
2. More complex
3. Requires understanding of hardware details (interfaces)

**Pass – 1 Assembler:**

An assembler does the following:

1. Generate machine instructions
   - evaluate the mnemonics to produce their machine code
   - evaluate the symbols, literals, addresses to produce their equivalent machine addresses
   - convert the data constants into their machine representations
2. Process pseudo operations

**Pass – 2 Assembler:**

A two-pass assembler performs two sequential scans over the source code:

*Pass 1*: symbols and literals are defined
*Pass 2*: object program is generated

*Parsing*: moving in program lines to pull out op-codes and operands

**Data Structures:**

- **Location counter (LC):** points to the next location where the code will be placed

- **Op-code translation table**: contains symbolic instructions, their lengths and their op-codes (or subroutine to use for translation)

- **Symbol table (ST):** contains labels and their values

- **String storage buffer (SSB):** contains ASCII characters for the strings

- **Forward references table (FRT):** contains pointer to the string in SSB and offset where its value will be inserted in the object code

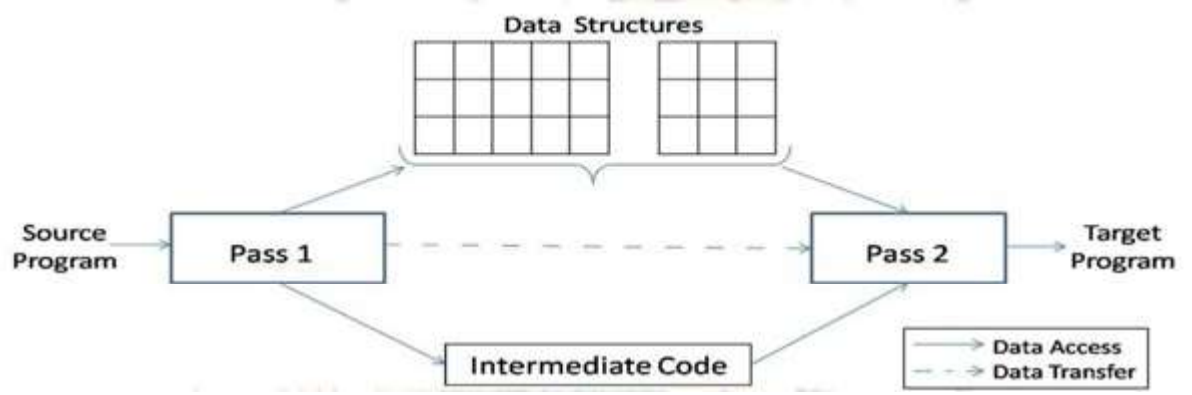



**Fig. 2 A Simple two pass assembler**

**Elements of Assembly Language :**

An assembly language programming provides three basic features which simplify programming when compared to machine language.

1. **Mnemonic Operation Codes :**
   Mnemonic operation code / Mnemonic Opcodes for machine instruction eliminate the need to memorize numeric operation codes. It enables assembler to provide helpful error diagnostics. Such as indication of misspelt operation codes.

2. **Symbolic Operands :**
   Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembler performs memory binding to these names; the programmer need not know any details of the memory bindings performed by the assembler.

3. **Data declarations :**
   Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example -5 into (11111010)2 or 10.5 into (41A80000)16

**Statement format :**

An assembly language statement has the following format :

[ Label] <Opcode> <operand Spec> [, operand Spec> ..]

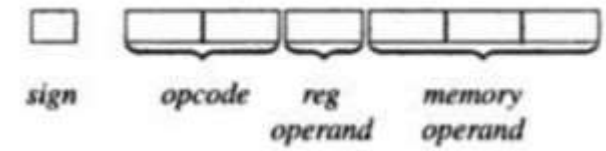
Where the notation [..] indicates that the enclosed specification is optional.

Label associated as a symbolic name with the memory word(s) generated for the statement

**Mnemonic Operation Codes :**

| *Instruction opcode* | *Assembly mnemonic* | *Remarks* |
|---|---|---|
| 00 | STOP | Stop execution |
| 01 | ADD | *First operand is modified* *Condition code is set* |
| 02 | SUB | |
| 03 | MULT | |
| 04 | MOVER | Register ← memory move |
| 05 | MOVEM | Memory ← register move |
| 06 | COMP | Sets condition code |
| 07 | BC | Branch on condition |
| 08 | DIV | Analogous to SUB |
| 09 | READ | *First operand is not used* |
| 10 | PRINT | |

**Instruction Format :**




Sign is not a part of Instruction

**An Assembly and equivalent machine language program :(solve it properly)**

```
          START   101
          READ    N               101)  + 09 0 113
          MOVER   BREG, ONE       102)  + 04 2 115
          MOVEM   BREG, TERM      103)  + 05 2 116
AGAIN     MULT    BREG, TERM      104)  + 03 2 116
          MOVER   CREG, TERM      105)  + 04 3 116
          ADD     CREG, ONE       106)  + 01 3 115
          MOVEM   CREG, TERM      107)  + 05 3 116
          COMP    CREG, N         108)  + 06 3 113
          BC      LT, AGAIN       109)  + 07 2 104
          MOVER   BREG, RESULT    110)  + 04 2 114
          PRINT   RESULT          111)  + 10 0 114
          STOP                    112)  + 00 0 000
N         DS      1               113)
RESULT    DS      1               114)
ONE       DC      '1'             115)  + 00 0 001
TERM      DS      1               116)
          END
```

**Assembly Language Statements :**

Three Kinds of Statements

a. Imperative Statements
b. Declaration Statements
c. Assembler Directives

**Imperative Statements :** It indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.

**Declaration Statements :** Two types of declaration statements is as follows

**[Label] DS <constant>**
**[Label] DC '<Value>'**

The DS (Declare Storage) statement reserves areas of memory and associates names with them.

Eg)**A DS 1**

**B DS 150**

First statement reserves a memory of 1 word and associates the name of the memory as A.

Second statement reserves a memory of 150 words and associates the name of the memory as B.

The DC (Declare Constant) Statement constructs memory word containing constants.

Eg ) **ONE DC '1'**

Associates the name ONE with a memory word containing the value '1'. The programmer can declare constants in decimal, binary, hexadecimal forms etc., These values are not protected by the assembler. In the above assembly language program the value of ONE Can be changed by executing an instruction MOVEM BREG,ONE

**Assembler Directives :**

Assembler directives instruct the assembler to perform certain actions during the assembly of a program. Some Assembler directives are described in the following

**START <Constant>**

Indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <Constant>

**END [ <operand spec>]**

It Indicates the end of the source program

**Pass Structure of Assembler :**

One complete scan of the source program is known as a pass of a Language Processor.

Two types 1) Single Pass Assembler 2) Two Pass Assembler.

**Single Pass Assembler :**

First type to be developed Most Primitive Source code is processed only once. The operand field of an instruction containing forward reference is left blank initially

Eg) MOVER BREG,ONE

Can be only partially synthesized since ONE is a forward reference during the scan of the source program, all the symbols will be stored in a table called **SYMBOL TABLE.** Symbol table consists of two important fields, they are symbol name and address.

All the statements describing forward references will be stored in a table called Table of Incomplete Instructions (TII)

**TII (Table of Incomplete instructions)**

| Instruction Address | Symbol |
|---|---|
| 101 | ONE |

By the time the END statement is processed the symbol table would contain the address of all symbols defined in the source program.

**Two Pass Assembler :**

Can handle forward reference problem easily.

*First Phase : (Analysis)*
Symbols are entered in the table called Symbol table
Mnemonics and the corresponding opcodes are stored in a table called Mnemonic table
LC Processing

*Second Phase : (Synthesis)*
Synthesis the target form using the address information found in Symbol table.
First pass constructs an Intermediated Representation (IR) of the source program for use by the second pass.

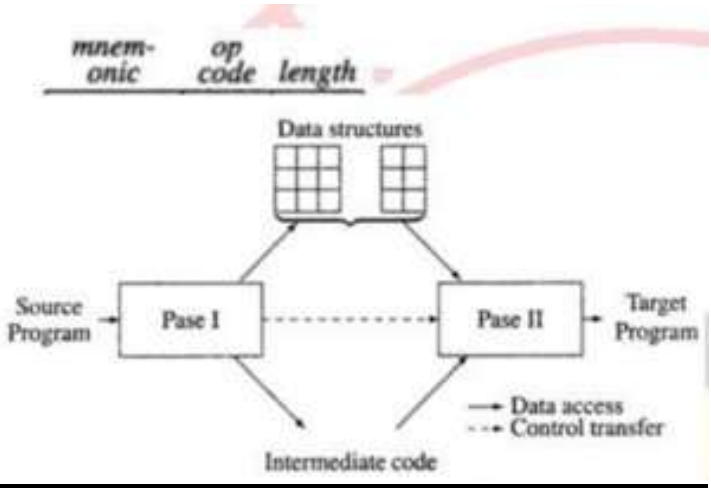
Data Structure used during Synthesis Phase :

1. Symbol table
2. Mnemonics table

| *symbol* | *address* |
|---|---|
| AGAIN | 104 |
| N | 113 |

| *mnem-onic* | *op code* | *length* |
|---|---|---|
| ADD | 01 | 1 |
| SUB | 02 | 1 |

Processed form of the source program called Intermediate Code (IC)




**ADVANCED ASSEMBLER DIRECTIVES**

- ORIGIN
- EQU
- LTROG

ORIGIN :

Syntax : ORIGIN < address spec>

<address spec>can be an <operand spec> or constant

Indicates that Location counter should be set to the address given by < address spec>

This statement is useful when the target program does not consist of consecutive memory words. Eg) ORIGIN Loop + 2

EQU : *Syntax*

<symbol> EQU <address spec>

<address spec>operand spec (or) constant

Simply associates the name symbol with address specification
No Location counter processing is implied

Eg ) Back EQU Loop

LTORG : (Literal Origin)

Where should the assembler place literals ?

It should be placed such that the control never reaches it during the execution of a program.

By default, the assembler places the literals after the END statement.

LTROG statement permits a programmer to specify where literals should be placed.

**Solve the below example.for Pass-1 Assembler.**

```
          START   101
          READ    X
          READ    Y
          MOVER   AREG, X
          ADD     AREG, Y
          MOVEM   AREG, RESULT
          PRINT   RESULT
          STOP
X         DS  1
Y         DS  1
RESULT    DS  1
          END
```

**Fig. 1.7.4 : A sample program for finding X + Y**

**Conclusion :**

Thus , I have implemented pass-1 assembler with symbol table, literal table and pool table.

# GROUP - A

## EXPERIMENT NO.: 02

**Title:**

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

**Objectives :**

- To understand Data structure of Pass-1 & Pass-2 assembler
- To understand Pass-1 & Pass-2 assembler concept
- To understand Advanced Assembler Directives

**Problem Statement :**

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment..

**Software Requirements:**

Latest jdk., Eclipse

**Theory**

**Concepts:**

**Introduction:-**

There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*. *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.

An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.


**Figure 1**. Executable program generation from an assembly source code

**Advantages of coding in assembly language are:**

- Provides more control over handling particular hardware components
- May generate smaller, more compact executable modules
- Often results in faster execution

**Disadvantages:**

- Not portable
- More complex
- Requires understanding of hardware details (interfaces)

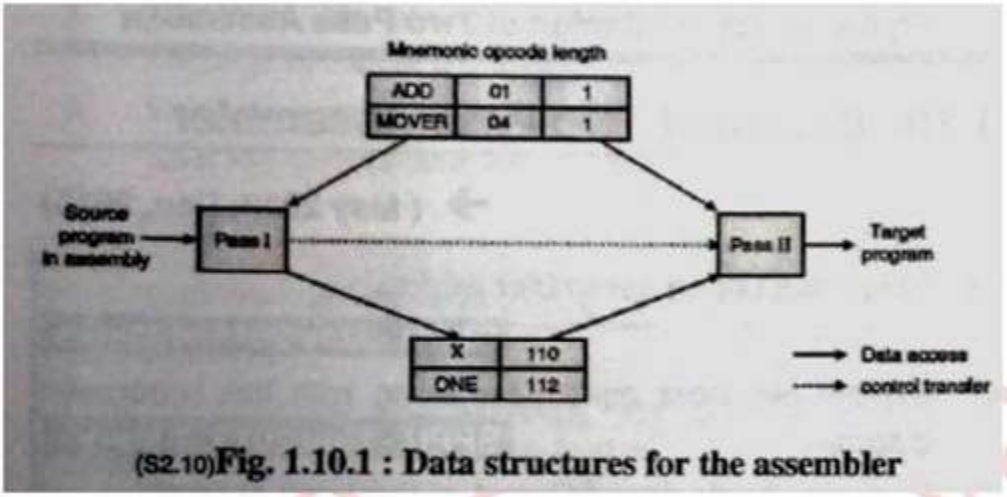
**DESIGN OF TWO PASS ASSEMBLER:**

Pass I : (Analysis of Source Program)

- Separate the symbol, mnemonic opcode and operand fields
- Build the symbol table.
- Perform LC processing.
- Construct intermediate representation

PASS 2:-

Processes the intermediate representation (IR) to synthesize the target program.




(S2.10)**Fig. 1.10.1 : Data structures for the assembler**

**Table 1.10.1 : An enhanced machine opcode table (MOT)**

| | Mnemonic opcode | Class | Opcode | Length |
|---|---|---|---|---|
| 0 | STOP | IS | 00 | 1 |
| 1 | ADD | IS | 01 | 1 |
| 2 | SUB | IS | 02 | 1 |
| 3 | MULT | IS | 03 | 1 |
| 4 | MOVER | IS | 04 | 1 |
| 5 | MOVEM | IS | 05 | 1 |
| 6 | COMP | IS | 06 | 1 |
| 7 | BC | IS | 07 | 1 |
| 8 | DIV | IS | 08 | 1 |
| 9 | READ | IS | 09 | 1 |
| 10 | PRINT | IS | 10 | 1 |
| 11 | START | AD | 01 | – |
| 12 | END | AD | 02 | – |
| 13 | ORIGIN | AD | 03 | – |
| 14 | EQU | AD | 04 | – |
| 15 | LTORG | AD | 05 | – |
| 16 | DS | DL | 01 | – |
| 17 | DC | DL | 02 | 1 |
| 18 | AREG | RG | 01 | – |
| 19 | BREG | RG | 02 | – |
| 20 | CREG | RG | 03 | – |
| 21 | | CC | 01 | – |

IS : Imperative Statement
AD : Assembler Directive
DL : Declaration Statement
RG : Register
CC : Comparison Condition

| | Mnemonic opcode | Class | Opcode | Length |
|---|---|---|---|---|
| 22 | LT | CC | 02 | – |
| 23 | GT | CC | 03 | – |
| 24 | LE | CC | 04 | – |
| 25 | GE | CC | 05 | – |
| 26 | NE | CC | 06 | – |
| 27 | ANY | CC | 07 | – |

**Note : solve below example in details**

```
          START   200
          MOVER   AREG, ='5'
          MOVEM   AREG, X
L1        MOVER   BREG, ='2'
          ORIGIN  L1+3
          LTORG
NEXT      ADD     AREG,='1'
          SUB     BREG,='2'
          BC      LT,BACK
          LTORG
BACK      EQU     L1
          ORIGIN  NEXT + 5
          MULT    CREG, 4
          STOP
X         DS      1
          END
```

**Conclusion :**

Thus , I have implemented Pass-2 assembler by taking input as output of assignment-1

# GROUP - B

## EXPERIMENT NO.: 3

**Title:**
Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

**Objectives :**
- To understand OS & SCHEDULLING Concepts
- To implement Scheduling FCFS, SJF, RR & Priority algorithms
- To study about Scheduling and scheduler

**Problem Statement :**
Write a Java program (using OOP features) to implement following scheduling algorithms: FCFS , SJF, Priority and Round Robin .

**Software Requirements:**
JDK/Eclipse

**Theory Concepts:**

**CPU Scheduling**:

CPU scheduling refers to a set of policies and mechanisms built into the operating systems that govern the order in which the work to be done by a computer system is completed.

- Scheduler is an OS module that selects the next job to be admitted into the system and next process to run.

- The primary objective of scheduling is to optimize system performance in accordance with the criteria deemed most important by the system designers.

**What is scheduling?**

Scheduling is defined as the process that governs the order in which the work is to be done. Scheduling is done in the areas where more no. of jobs or works are to be performed. Then it requires some plan i.e. scheduling that means how the jobs are to be performed i.e. order. CPU scheduling is best example of scheduling.

**What is scheduler?**

1. Scheduler in an OS module that selects the next job to be admitted into the system and the next process to run.
2. Primary objective of the scheduler is to optimize system performance in accordance with the criteria deemed by the system designers. In short, scheduler is that module of OS which schedules the programs in an efficient manner.

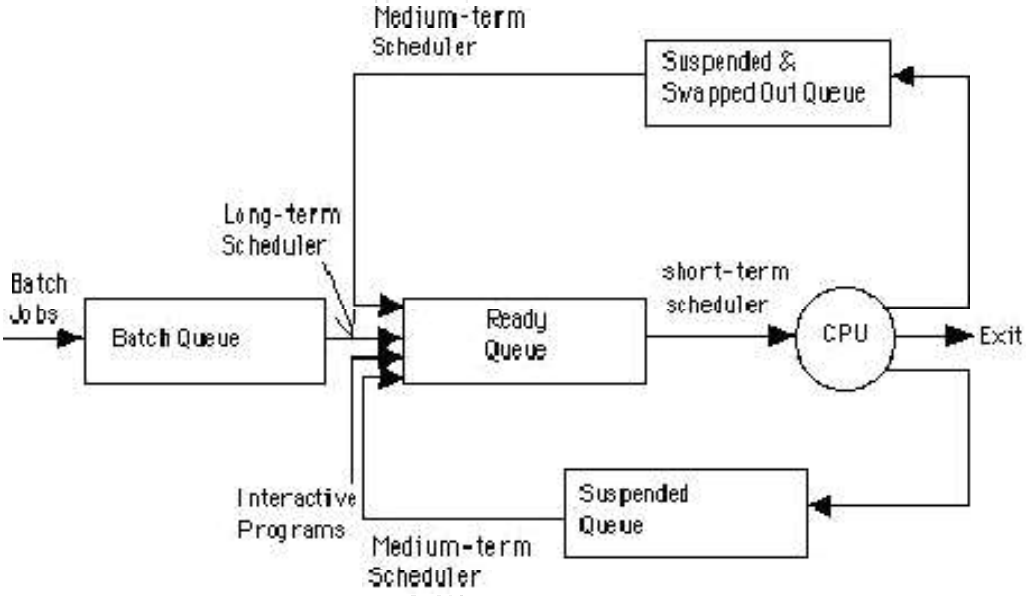
**Necessity of scheduling**

- Scheduling is required when no. of jobs are to be performed by CPU.
- Scheduling provides mechanism to give order to each work to be done.
- Primary objective of scheduling is to optimize system performance.
- Scheduling provides the ease to CPU to execute the processes in efficient manner.

**Types of schedulers**

In general, there are three different types of schedulers which may co-exist in a complex operating system.

- Long term scheduler
- Medium term scheduler
- Short term scheduler.




**Long Term Scheduler**

• The long term scheduler, when present works with the batch queue and selects the next batch job to be executed.

• Batch is usually reserved for resource intensive (processor time, memory, special I/O devices) low priority programs that may be used fillers of low activity of interactive jobs.

- Batch jobs usually also contains programmer-assigned or system-assigned estimates of their resource needs such as memory size, expected execution time and device requirements.

- Primary goal of long term scheduler is to provide a balanced mix of jobs.

**Medium Term Scheduler**

• After executing for a while, a running process may because suspended by making an I/O request or by issuing a system call.

• When number of processes becomes suspended, the remaining supply of ready processes in systems where all suspended processes remains resident in memory may become reduced to a level that impairs functioning of schedulers.

3. The medium term scheduler is in charge of handling the swapped out processes.
4. It has little to do while a process is remained as suspended.

**Short Term Scheduler**

- The short term scheduler allocates the processor among the pool of ready processes resident in the memory.
- Its main objective is to maximize system performance in accordance with the chosen set of criteria.
- Some of the events introduced thus for that cause rescheduling by virtue of their ability to change the
- global system state are:
- Clock ticks
- Interrupt and I/O completions
- Most operational OS calls
- Sending and receiving of signals
- Activation of interactive programs.
- Whenever one of these events occurs ,the OS involves the short term scheduler.

**Scheduling Criteria :**

- **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.

**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.

**Types of scheduling Algorithms**

- In general, scheduling disciplines may be pre-emptive or non-pre-emptive .
- In batch, non-pre-emptive implies that once scheduled, a selected job turns to completion. There are different types of scheduling algorithms such as:
    - FCFS(First Come First Serve)
    - SJF(Short Job First)
    - Priority scheduling
    - Round Robin Scheduling algorithm

**First Come First Serve Algorithm**

- FCFS is working on the simplest scheduling discipline.
- The workload is simply processed in an order of their arrival, with no pre-emption.
- FCFS scheduling may result into poor performance.
- Since there is no discrimination on the basis of required services, short jobs may considerable in turn around delay and waiting time.

**Advantages**

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

**Disadvantages**

- Convoy effect occurs. Even very small process should wait for its turn to come to utilize the CPU. Short process behind long process results in lower CPU utilization.
- Throughput is not emphasized.

## First Come, First Served

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

- **Suppose that the processes arrive in the order: *P1* , *P2* , *P3***
- **The Gantt Chart for the schedule is:**


- **Waiting time for *P1* = 0; *P2* = 24; *P3* = 27**
- **Average waiting time: (0 + 24 + 27)/3 = 17**

**Shortest Job First Algorithm :**

- his is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

**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.
- Starvation may be possible for the longer processes.

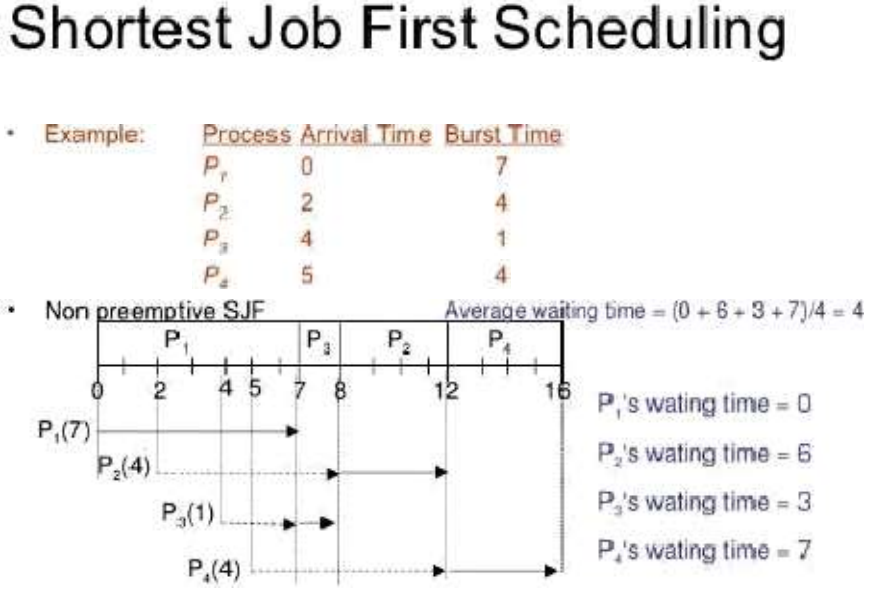
**This algorithm is divided into two types:**

Pre-emptive SJF

Non-pre-emptive SJF

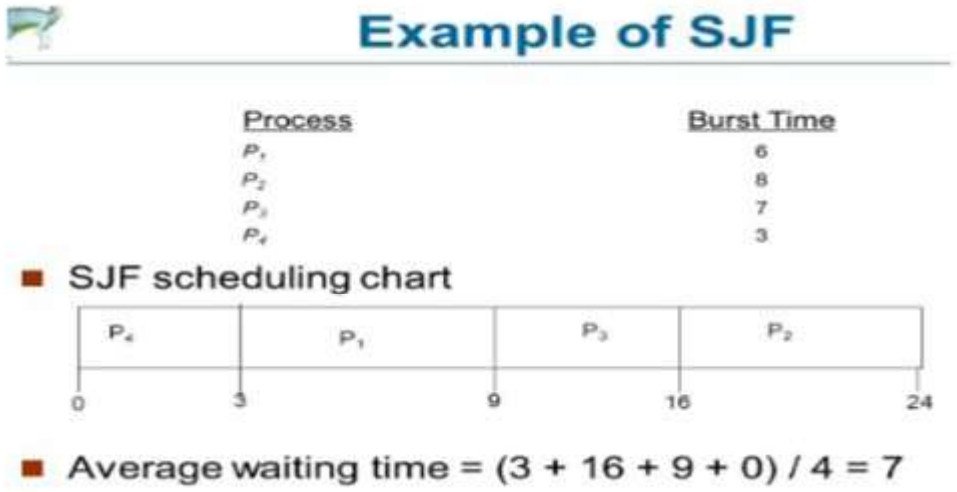
**Pre-emptive SJF Algorithm:**

In this type of SJF, the shortest job is executed 1st. the job having least arrival time is taken first for execution. It is executed till the next job arrival is reached.




**Non-pre-emptive SJF Algorithm:**

In this algorithm, job having less burst time is selected 1st for execution. It is executed for its total burst time and then the next job having least burst time is selected.




## Round Robin Scheduling :

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processe

**Advantages**

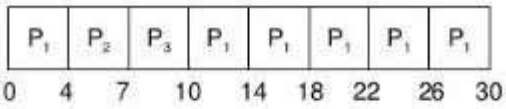
- Round-robin is effective in a general-purpose, times-sharing system or transaction-processing system.
- Fair treatment for all the processes.
- Overhead on processor is low.
- Overhead on processor is low.
- Good response time for short processes.

**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.

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

- **Quantum time = 4 milliseconds**
- **The Gantt chart is:**




- **Average waiting time = {[0+(10-4)]+4+7}/3 = 5.6**

## Priority Scheduling :

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

**Advantage**

- Good response for the highest priority processes.

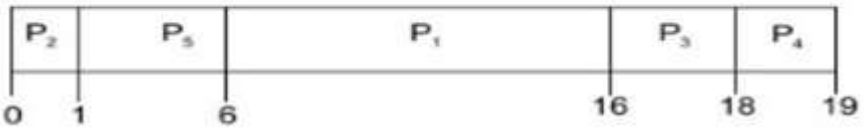
**Disadvantage**

Starvation may be possible for the lowest priority processes

# Priority

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

- **Gantt Chart**




- **Average waiting time = (6 + 0 + 16 + 18 + 1)/5 = 8.2**

**Algorithms(procedure) :**

**FCFS :**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+

Burst time for process(n) Step 6: Calculate

(a) Average waiting time = Total waiting Time / Number of process

(b) Average Turnaround time = Total Turnaround Time /

Number of process Step 7: Stop the process

**SJF :**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to
highest burst time.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(c) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(d) Turn around time for Process(n)= waiting time of Process(n)+ Burst time

for process(n) Step 6: Calculate

(c) Average waiting time = Total waiting Time / Number of process

(d) Average Turnaround time = Total Turnaround Time /

Number of process Step 7: Stop the process

**RR :**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where

No. of time slice for process(n) = burst time process(n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

(a) Waiting time for process(n) = waiting time of process(n-1)+ burst time of process(n-1 ) + the time difference in getting the CPU from process(n-1)

(b) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+

the time difference in getting CPU from process(n).

Step 7: Calculate

(e) Average waiting time = Total waiting Time / Number of process

(f) Average Turnaround time = Total Turnaround Time /

Number of process Step 8: Stop the process.

**Priority Scheduling :**

**Algorithms :**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time, priority

Step 4: Start the Ready Q according the priority by sorting according

to lowest to highest burst time and process.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

(e) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(f) Turn around time for Process(n)= waiting time of Process(n)+ Burst time

for process(n) Step 6: Calculate

(g) Average waiting time = Total waiting Time / Number of process

(h) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

**Conclusion:**

Hence we have studied that-

- CPU scheduling concepts like context switching, types of schedulers, different timing parameter like waiting time, turnaround time, burst time, etc.
- Different CPU scheduling algorithms like FIFO, SJF,Etc.
- FIFO is the simplest for implementation but produces large waiting times and reduces system performance.
- SJF allows the process having shortest burst time to execute first.

# EXPERIMENT NO.: 4

**Title:**

Write a java program to implement Page Replacement Policies LRU & OPT.

**Objectives :**

- To understand Page replacement policies
- To understand paging concept
- To understand Concept of page fault, page hit, miss, hit ratio etc

**Problem Statement:**

Write a java program to implement Page Replacement Policies LRU & OPT.

**Software Requirements:**

Latest jdk., Eclipse

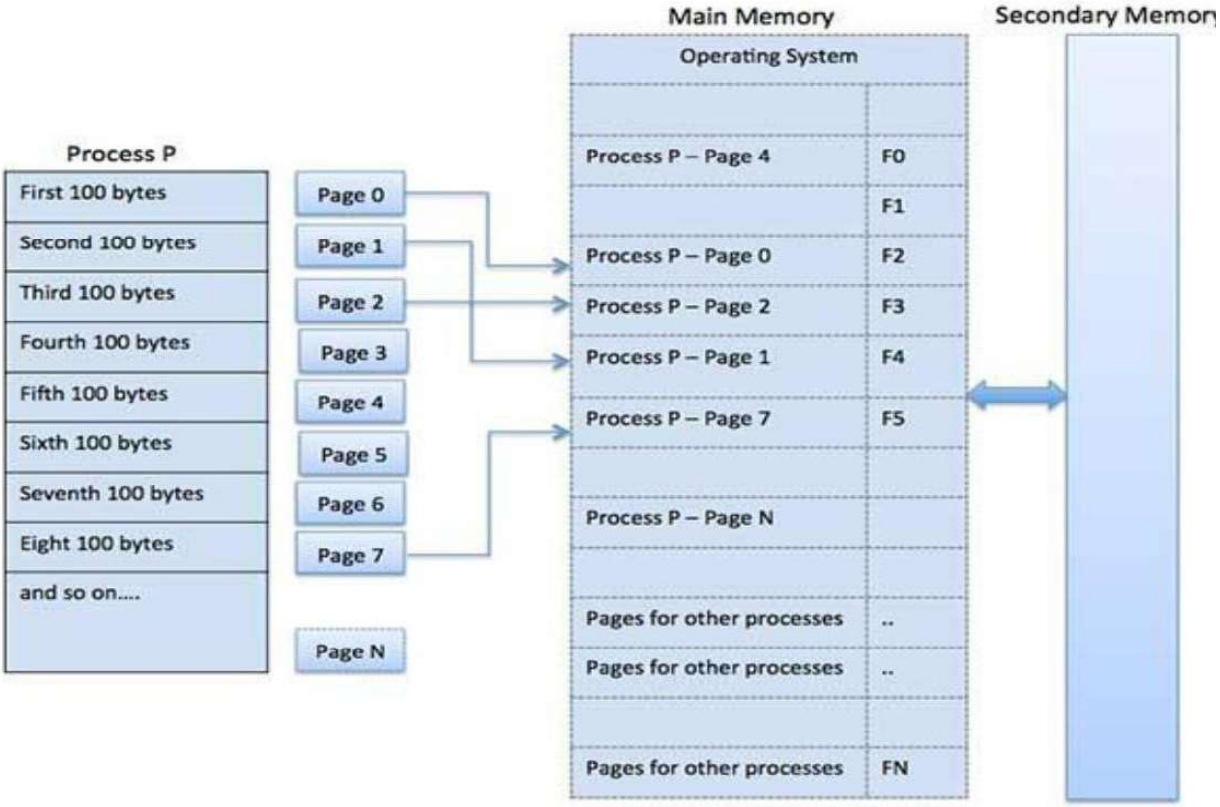
**Theory Concepts:**

**Paging :**

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.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.




Address Translation

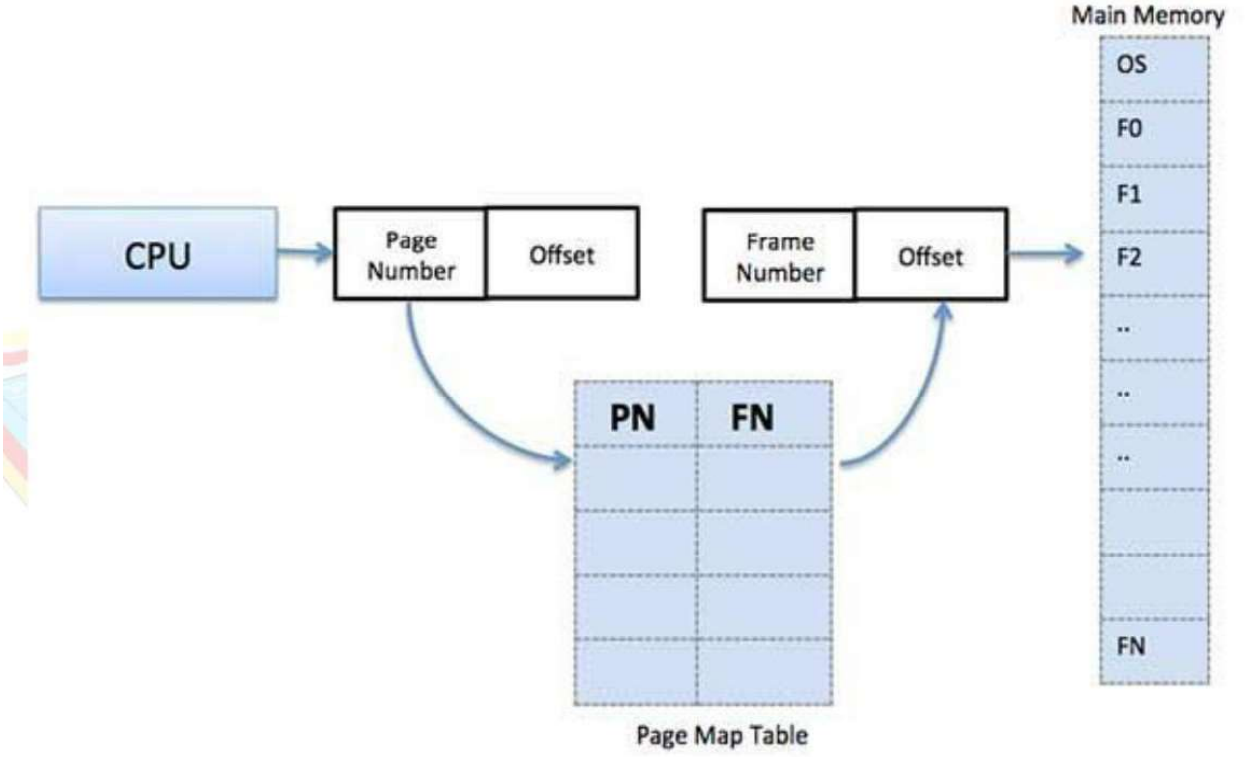
Page address is called **logical address** and represented by **page number**and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.




When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.
When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.
This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

**Advantages and Disadvantages of Paging**

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.

- Page table requires extra memory space, so may not be good for a system having small RAM.

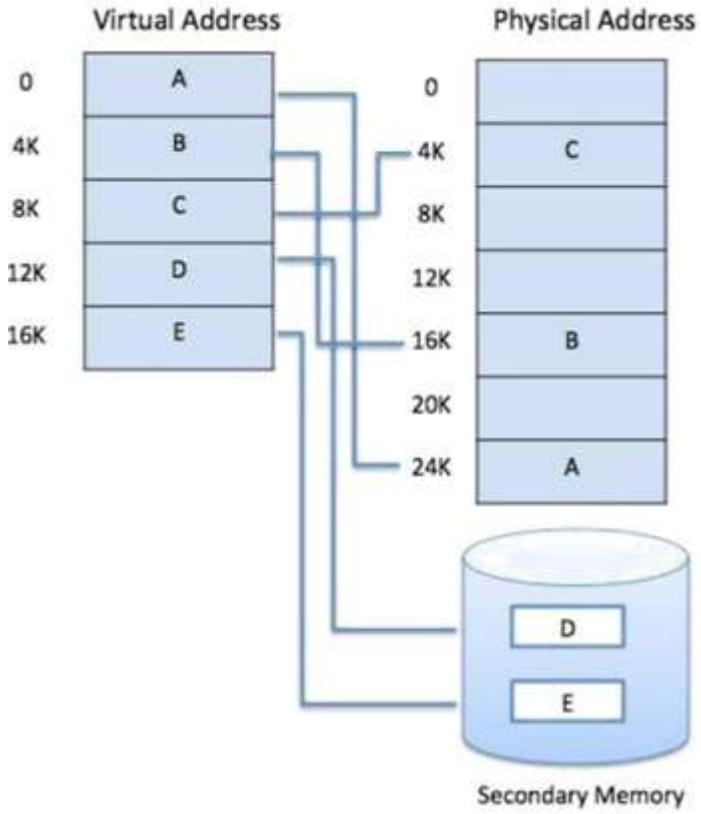
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

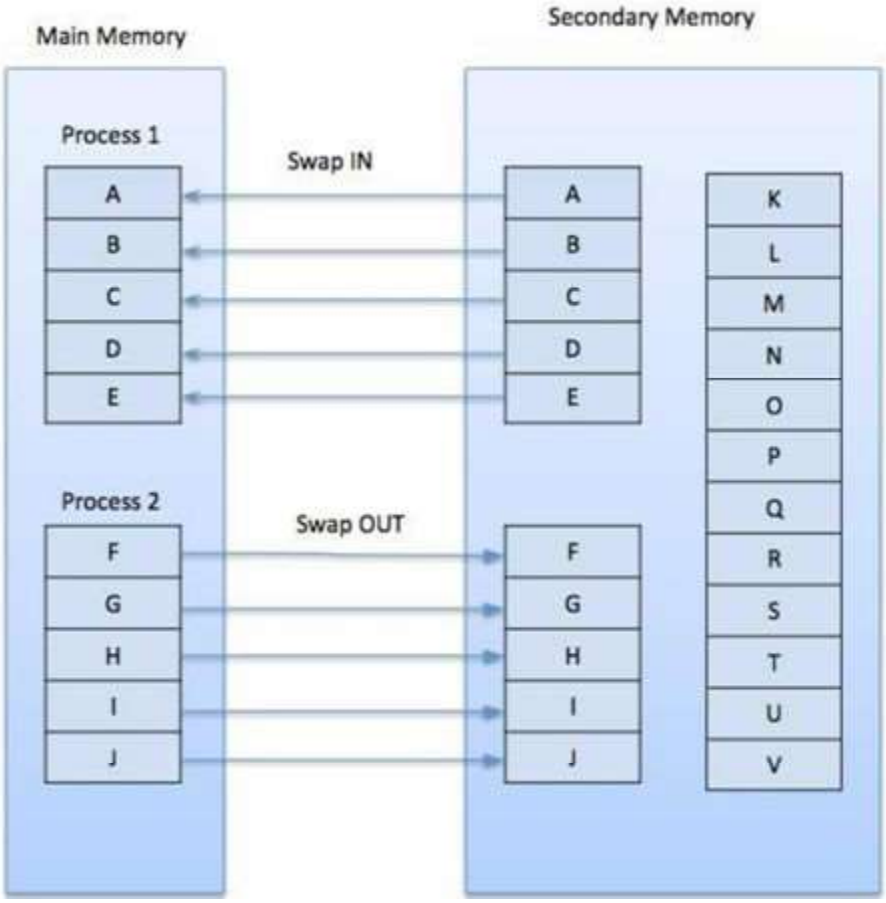
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −




Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

**Demand Paging**

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.




While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

**Advantages**

Following are the advantages of Demand Paging −

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

**Disadvantages**

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

**Page Replacement Algorithm :**

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

**Page fault :**

A **page fault** (sometimes called #PF, PF or hard **fault**) is a type of exception raised by computer hardware when a running program accesses a memory **page**that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.

**Page hit :**

A **hit** is a request to a web server for a file, like a web **page**, image, JavaScript, or Cascading Style Sheet. When a web **page** is downloaded from a server the number of "**hits**" or "**page hits**" is equal to the number of files requested.

**Page frame :**

The **page frame** is the storage unit (typically 4KB in size) whereas the **page** is the contents that you would store in the storage unit ie the **page frame**. For eg) the RAM is divided into fixed size blocks called **page frames** which is typically 4KB in size, and each **page frame** can store 4KB of data ie the **page**.

**Page table :**

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

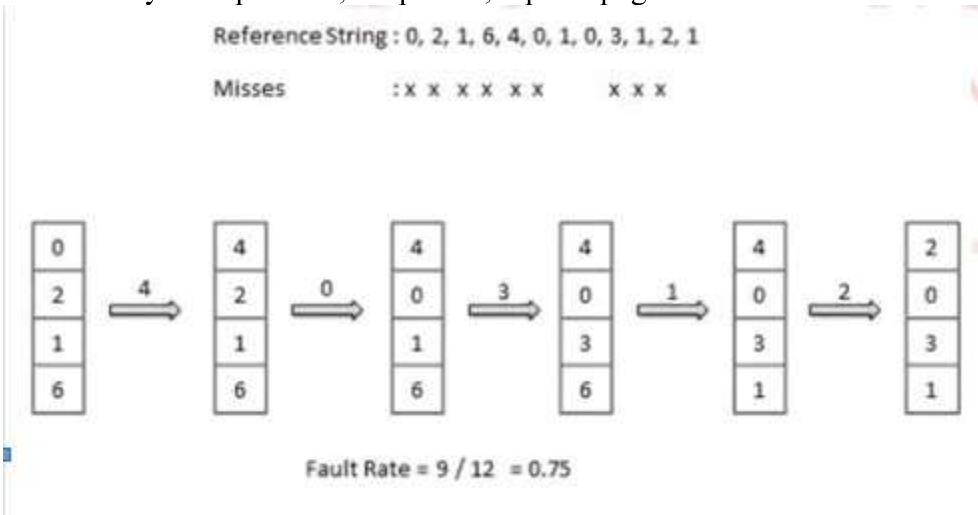
**Reference String :**

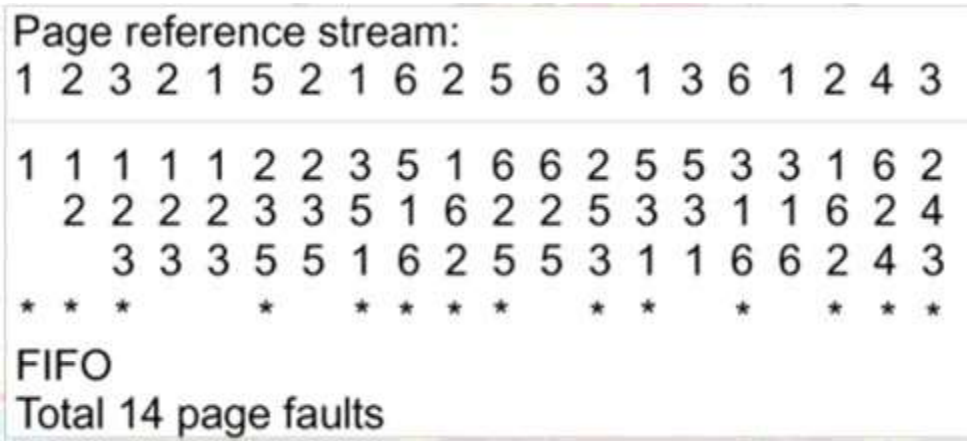
The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96

**First In First Out (FIFO) algorithm :**

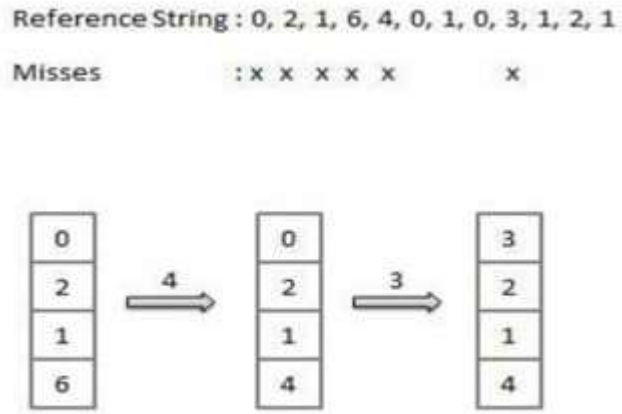
- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.



**Optimal Page algorithm :**

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

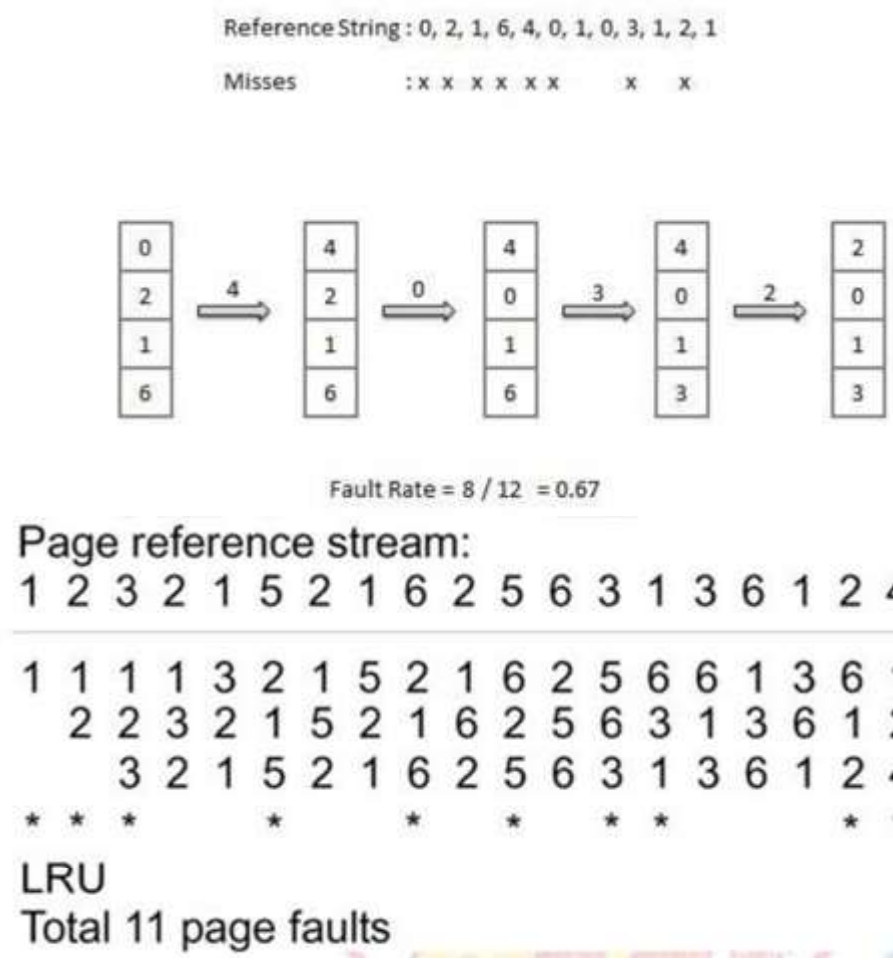




**Note : you can take other example also. This just for reference. (you must calculate page fault, page hit and hit ratio )**

**Least Recently Used (LRU) algorithm :**

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.




**Note : you can take other example also. This just for reference. (you must calculate page fault, page hit and hit ratio )**

**Page Buffering algorithm**

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

**Least frequently Used(LFU) algorithm**

- The page with the smallest count is the one which will be selected for replacement.

- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

**Most frequently Used(MFU) algorithm**

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

**Conclusion :**

Thus , I have implemented page replacement policies `LRU and OPT.