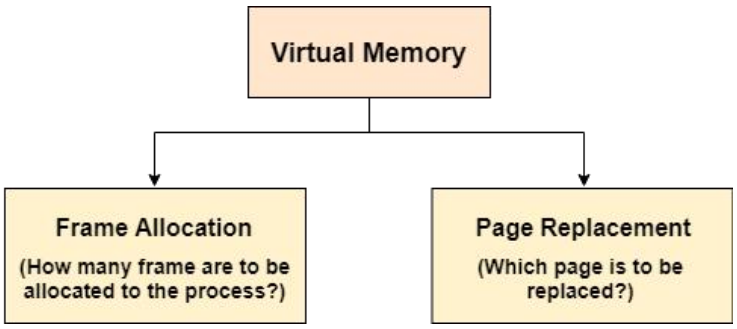
Assignment No. 7 Write a program to simulate Page replacement algorithm.

Theory:

Virtual memory uses both hardware and software to enable a computer to compensate for main memory shortages, temporarily transferring data from random access memory (RAM) to disk storage. Computers have a finite amount of RAM, so memory will eventually run out when multiple programs run at the same time. A system using virtual memory uses a section of the hard drive to emulate RAM. With virtual memory, a system can load larger or multiple programs running at the same time, enabling each one to operate as if it has more space, without having to purchase more RAM.



There are two main aspects of virtual memory, Frame allocation and Page Replacement. It is very important to have the optimal frame allocation and page replacement algorithm. Frame allocation is all about how many frames are to be allocated to the process while the page replacement is all about determining the page number which needs to be replaced in order to make space for the requested page.

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

FIFO - This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

LRU - In this algorithm page will be replaced which is least recently used.

OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

Algorithm for FIFO:

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the pages in the queue to perform FIFO.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Remove the first page from the queue as it was the first to be entered in the memory

b) Replace the first page in the queue with the current page in the string.

c) Store current page in the queue.

d) Increment page faults.

2. Return page faults.

Algorithm for LRU:

Let **capacity** be the number of pages that memory can hold. Let **set** be the current set of pages in memory.

1- Start traversing the pages.
i) **If set holds less pages than capacity.**
a) Insert page into the set one by one until the size of **set** reaches **capacity** or all page requests are processed.
b) Simultaneously maintain the recent occurred index of each page in a map called **indexes**.
c) Increment page fault
ii) **Else**
If current page is present in **set**, do nothing.
Else
a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.
2. Return page faults.

Algorithm for Optimal:

1. Push the first page in the memory as per the memory demand.
2. Push the second page as per the memory demand.
3. Push the third page until the memory is full.
4. As the queue is full, the page which is least recently used is popped.
5. Repeat step 4 until the page demand continues and until the processing is over.
6. Terminate the program.

Conclusion: Hence we have successfully simulated Page Replacement Algorithms.

Assignment No 06: Write a program to simulate Memory placement strategies for Best fit, First fit, Next fit,Worst fit.

1.1 Prerequisite:

Basic concepts of Memory Management, Different types of Memory Management.

1.2 Theory:

- **Memory Management**

  Memory management is the process of controlling and coordinating a computer's main memory. It ensures that blocks of memory space are properly managed and allocated so the operating system (OS), applications and other running processes have the memory they need to carry out their operations.

  Memory management takes into account the capacity limitations of the memory device itself, deallocating memory space when it is no longer needed or extending that space through virtual memory. Memory management strives to optimize memory usage so the CPU can efficiently access the instructions and data it needs to execute the various processes.

- **Memory Allocation**

  Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space.

  Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. Memory allocation is achieved through a process known as memory management.
  There are four memory allocation algorithms:
  1. Single contiguous allocation.
  2. Partitioned allocation.
  3. Paged memory management.
  4. Segmented memory management.

- **Partitioning Allocation:**

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

- **Partitioning Allocation Algorithm:**

A partition allocation method is considered better if it avoids internal fragmentation. When its time to load a process into the main memory and if there is more than one free block of memory of sufficient size then the OS decides which free block to allocate.

There are four algorithms which are implemented by the operating system in order to find out the holes in the linked list and allocate them to processes.

The explanation about each of the algorithm is given below:

1. First fit Algorithm:

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

- **ADVANTAGES:**

  Fastest algorithm because it searches as little as possible.

- **DISADVANTAGES:**

  The remaining unused memory areas left after allocation become waste if it is too smaller. Thus, request for larger memory requirement cannot be accomplished.

- **ALGORITHM:**

  Step no 01: Get number of processes and number of blocks.
  Step no 02: After that get the size of each block and process requests.
  Step no 03: Now allocate processes
  if (block size >= process size)
  // allocate the process
  Else
  //move on to next block
  Step no 04: Display the processes with the blocks that are allocated to a respective process.
  Step no 05: Stop.

2. Best fit Algorithm:

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

- **ADVANTAGES:**

  Memory utilization is much better than first fit as it searches the smallest free partition first available.

- **DISADVANTAGES:**
  It is slower and may even tend to fill up memory with tiny useless holes.

- **ALGORITHM:**
  Step 1: Get number of processes and number of blocks.
  Step 2: After that get the size of each block and process requests.
  Step 3: Then select the best memory block that can be allocated using the above definition.
  Step 4: Display the processes with the blocks that are allocated to a respective process.
  Step 5: Value of fragmentation is optional to display to keep track of wasted memory.
  Step 6: Stop.

3. Worst fit Algorithm:

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

- **ADVANTAGES:**
  Reduces the rate of production of small gaps.

- **DISADVANTAGES:**
  It is a process requiring larger memory arrives at a larger stage then it cannot be accommodated as the largest hole is already split and occupied.

- **ALGORITHM:**
  Step no 01: Input memory block with a size.
  Step no 02: Input process with size.
  Step no 03: Initialize by selecting each process to find the maximum block size that can be assigned to the current process.
  Step no 04: If the condition does not fulfil, they leave the process.
  Step no 05: If the condition is not fulfilled, then leave the process and check for the next process.
  Step no 06: Stop.

4. Next fit Algorithm:

Next fit is a modified version of first fit. It begins as first fit to find a free partition. When called next it starts searching from where it left off, not from the beginning.

- **ADVANTAGES:**
  Next fit tries to address this problem by starting the search for the free portion of parts not from the start of the memory, but from where it ends last time.

Next fit is a very fast searching algorithm and is also comparatively faster than First fit and Best fit Memory management algorithms.

- **DISADVANTAGES:**
  Next fit does not scan the whole list, it starts scanning the list from the next node. The idea behind the next fit is the fact that the list has been scanned once therefore the probability of finding the hole is larger in the remaining part of the list.

- **ALGORITHM:**
  Step no 01: Enter the number of memory blocks.
  Step no 02: Enter the size of each memory block.
  Step no 03: Enter the number of processes with their sizes.
  Step no 04: Start by selecting each process to check if it can be assigned to the current memory block.
  Step no 05: If the condition in step 4 is true, then allocate the process with the required memory and check for the next process from the memory block where searching was halted, not from the starting.
  Step no 06: If the current memory size is smaller, then continue to check the next blocks.
  Step no 07: Stop.

CONCLUSION: Hence we successfully designed First fit algorithm, Best fit algorithm, Worst time algorithm, Next time algorithm.

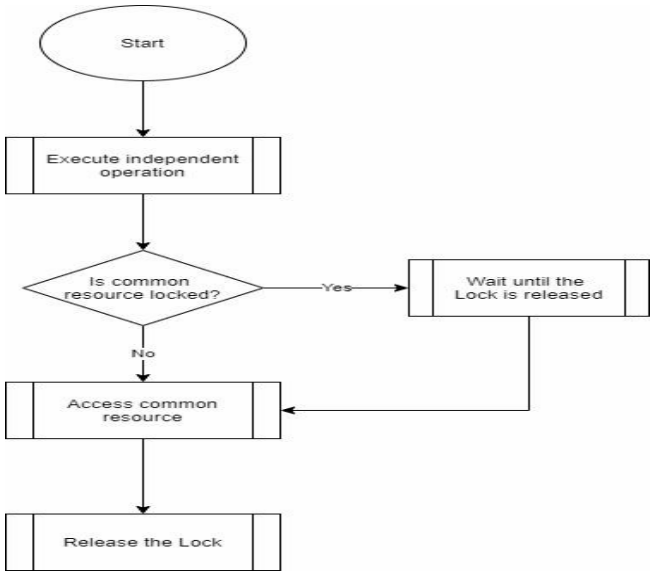
Assignment No. 5: Write a program to solve Classical Problems of Synchronization usingMutex and Semaphore.

Theory:

Mutex

- ✓ Mutex is used to ensure only one thread at a time can access the resource protected bythe mutex. The process that look the mutex must be the one that unlock it. Mutex is good only for managing mutual exclusion to some shared resource.
- ✓ Mutex is easy and efficient for implement.
- ✓ Mutex can be in one of two states: locked or unlocked.
- ✓ Mutex is represented by one bit. Zero (0) means unlock and other value represents locked. It uses two procedures.
- ✓ When a process need access to a critical section, it checks the condition of mutex_locks. If the mutex is currently unlocked then calling process enters into critical section.
- ✓ If the mutex is locked, the calling process is entered into blocked state and wait until the process in the critical section finishes its execution.
- ✓ Mutex variable have only two states so they are simple implement. Their use is limited to guarding entries to critical resigns.
- ✓ Mutex variable is like a binary semaphore. But both are not same.

Algorithm:



Semaphore :

- ✓ Semaphore is described by Dijkstra. Semaphore is a non-negative integer variable thatis used as a flag. Semaphore is an operating system abstract data type. If takes only integer value. It is used to solve critical section problem.
- ✓ Dijkstra introduces two operations (p and v) to operate on semaphore to solve process synchronization problem. A process calls the p operation when it wants to enter its critical section and calls v operation when it when to exit its critical section. The p operation and is called as signal operation.
- ✓ A wait operation on a semaphore decreases its value by one.

  Waits : < 0
  Do loops;
  S: = S – 1;
- ✓ A signal operation increments its value:
  Signal:
  - S := S + 1;
- ✓ A proper semaphore implementation requires that p and v indivisible operations. A semaphore operation is atomic. This may be possible by taking hardware support. The operation p and v are executed by the operating system in response to calls issued by any one process naming a semaphore as parameter.
- ✓ There is no guarantee that no two processes can execute wait and signal operations onthe same semaphore at the same time.

Properties of semaphore :

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Semaphore acquire many resources simultaneously.

Types of Semaphores :

There are mainly two types of Semaphores, or two types of signaling integer variables:

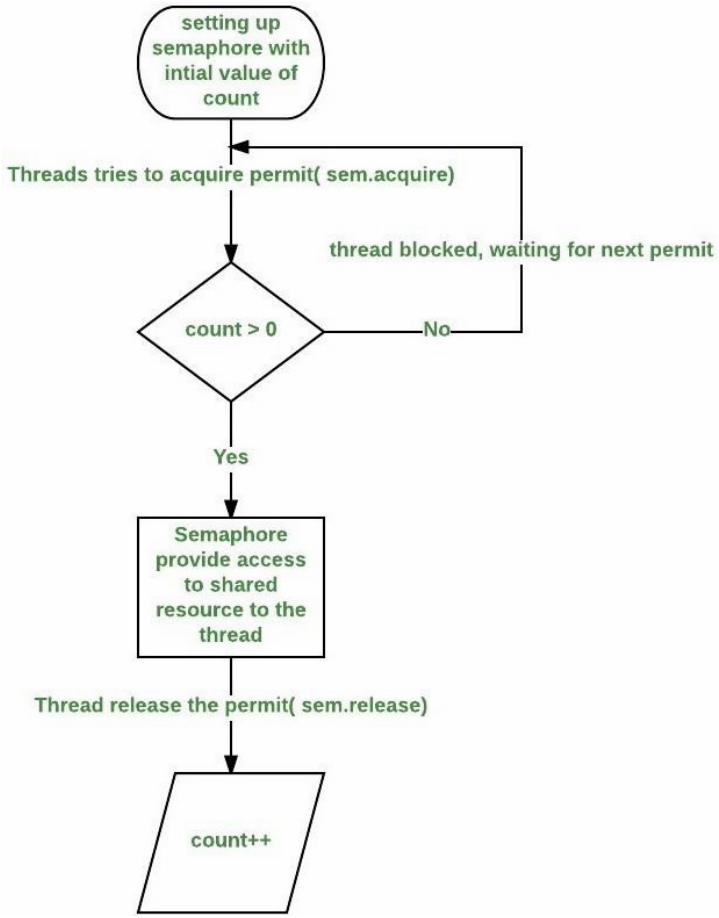
Binary Semaphores :

In this type of Semaphores the integer value of the semaphore can only be either 0 or 1. If the value of the Semaphore is 1, it means that the process can proceed to the critical section (the common section that the processes need to access). However, if the value of binary semaphore is 0, then the process cannot continue to the critical section of the code. When a process is using the critical section of the code, we change the Semaphore value to 0, and when a process is not using it, or we can allow a process to access the critical section, we change the value of semaphore to 1. Binary semaphore is also called mutex lock.

Counting Semaphores :

Counting semaphores are signaling integers that can take on any integer value. Using these Semaphores we can coordinate access to resources and here the Semaphore count is the number of resources available. If the value of the Semaphore is anywhere above 0, processes can access the critical section, or the shared resources. The number of processes that can access the resources / code is the value of the semaphore.
However, if the value is 0, it means that there aren't any resources that are available or the critical section is already being accessed by a number of processes and cannot be accessedby more processes.

Algorithm:



Process Synchronization Problems are as follows:

1. **Producer Consumer Problem**
2. **Reader-Writer Problem**
3. **Dining Philosopher Problem**

1. **Producer Consumer Problem:**

Consider a fixed-size buffer shared between a producer and a consumer.

- The producer generates an item and places it in the buffer.
- The consumer removes an item from the buffer.

The buffer is the critical section. At any moment:

- A producer cannot place an item if the buffer is full.
- A consumer cannot remove an item if the buffer is empty.

To manage this, we use three semaphores:

- mutex – ensures mutual exclusion when accessing the buffer.
- full – counts the number of filled slots in the buffer.
- empty – counts the number of empty slots in the buffer.

Semaphore Initialization

```
mutex = 1; // binary semaphore for mutual exclusion
full = 0; // initially no filled slots
empty = n; // buffer size
```

Producer

```
do {
// Produce an item
wait(empty); // Check for empty slot
wait(mutex); // Enter critical section
// Place item in buffer
signal(mutex); // Exit critical section
signal(full); // Increase number of full slots
} while (true);
```

Consumer

```
do {
wait(full); // Check for filled slot
wait(mutex); // Enter critical section
// Remove item from buffer
signal(mutex); // Exit critical section
signal(empty); // Increase number of empty slots
} while (true);
```

Explanation:

- Empty ensures that producers don't overfill the buffer.
- Full ensures that consumers don't consume from an empty buffer.
- Mutex ensures mutual exclusion, so only one process accesses the buffer at a time.

2. **Reader-Writer Problem**

The Readers-Writers Problem is a classic synchronization issue in operating systems. It deals with coordinating access to shared data (e.g., database, file) by multiple processes or threads.

- **Readers:** Multiple readers can read the shared data simultaneously without causing inconsistency (since they don't modify data).
- **Writers:** Only one writer can access the data at a time, and no readers are allowed while writing (to prevent data corruption).

The challenge is to design a synchronization scheme that ensures:

1. Multiple readers can access data together if no writer is writing.
1. Writers have exclusive access no other reader or writer can enter during writing.

Variants of the Problem

Readers Preference

- Readers are given priority.
- No reader waits if the resource is available for reading, even if a writer is waiting.
- Writers may suffer from *starvation*.

Writers Preference

- Writers are prioritized over readers.
- Ensures writers won't starve, but readers may wait longer.

Solution When Reader Has the Priority Over Writer

Here priority means, no reader should wait if the shared resource is currently open for reading. There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Reader Process (Reader Preference)

1. Reader requests the entry to critical section.
2. If allowed:

- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {
wait(mutex); // Lock before updating readcnt
readcnt++;
if (readcnt == 1)
wait(wrt); // First reader blocks writers
signal(mutex); // Allow other readers in
// ---- Critical Section (Reading) ----
wait(mutex);
readcnt--;
if (readcnt == 0)
signal(wrt); // Last reader allows writers
signal(mutex); // Unlock
} while(true);
```

Explanation:

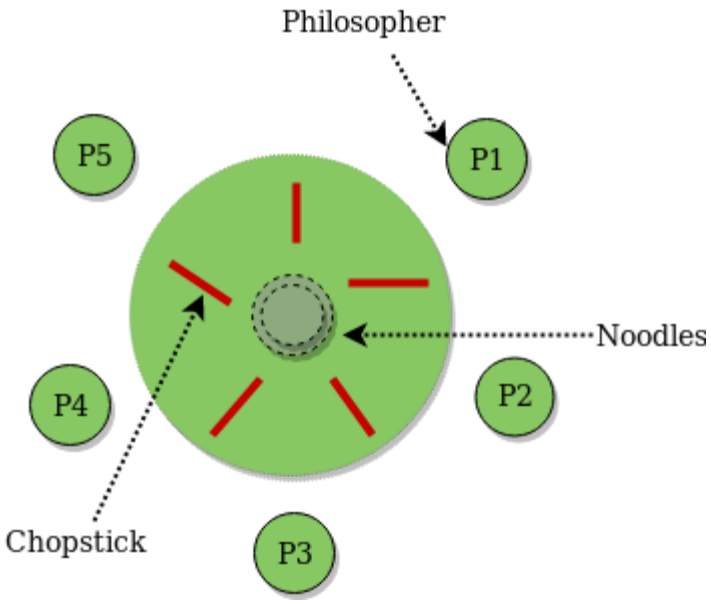
- When a reader enters, it locks mutex to update readcnt.
- If it's the first reader, it locks wrt so writers are blocked.
- Multiple readers can now read the data simultaneously.
- When a reader exits, it decrements readcnt.
- If it's the last reader, it unlocks wrt so writers can proceed.
  The first reader blocks writers, the last reader allows writers, and all readers in between share the resource. This gives preference to readers, but writers may starve.
  Writer's Process
  do {
  wait(wrt); // Lock resource

  // ---- Critical Section (Writing) ----

  signal(wrt); // Release resource
  } while(true);
  Explanation:
- **wait(wrt):** Writer locks the resource to get exclusive access.
- **Critical Section:** Writer performs writing (no other reader or writer can enter).
- **signal(wrt):** Writer releases the resource after finishing.
  Only one writer can write at a time, and no readers are allowed while writing. This ensures data integrity.
  The Readers-Writers Problem highlights the need for proper synchronization when multiple processes access shared data.
- Readers Preference solution allows many readers to access simultaneously while blocking writers, improving read efficiency.
- However, writers may starve if readers keep arriving continuously.
- To avoid this, Writers Preference or a Fair solution (using queues or advanced semaphores) can be used to ensure no starvation and balanced access.

3. Dining Philosopher Problem

The Dining Philosopher Problem is a classic synchronization problem introduced by Edsger Dijkstra in 1965. It illustrates the challenges of resource sharing in concurrent programming, such as deadlock, starvation, and mutual exclusion.



Problem Statement

- K philosophers sit around a circular table.
- Each philosopher alternates between thinking and eating.
- There is one chopstick between each philosopher (total K chopsticks).
- A philosopher must pick up two chopsticks (left and right) to eat.
- Only one philosopher can use a chopstick at a time.

The challenge: Design a synchronization mechanism so that philosophers can eat without causing **deadlock** (all waiting forever) or **starvation** (some never get a chance to eat).

Issues in the Problem

1. **Deadlock:** If every philosopher picks up their left chopstick first, no one can pick up the right one circular wait.
1. **Starvation:** Some philosophers may never get a chance to eat if others keep eating.
1. **Concurrency Control:** Must ensure no two adjacent philosophers eat simultaneously.

Semaphore Solution to Dining Philosopher

We use **semaphores** to manage chopsticks and avoid deadlock.

Algorithm

- Each chopstick is represented as a binary semaphore (mutex).
- Philosopher must acquire both left and right semaphores before eating.
- After eating, the philosopher releases both semaphores.

Pseudocode

```
semaphore chopstick[5] = {1,1,1,1,1};
Philosopher(i):
while(true) {
think();
wait(chopstick[i]); // pick left chopstick
wait(chopstick[(i+1)%5]); // pick right chopstick
eat();
signal(chopstick[i]); // put left chopstick
signal(chopstick[(i+1)%5]); // put right chopstick
}
```

Explanation:

- **semaphore chopstick[5] = {1,1,1,1,1};** Each chopstick is a binary semaphore initialized to

1 (available).

- **think();** Philosopher spends time thinking.
- **wait(chopstick[i]);** Tries to pick the left chopstick. If it's free, philosopher takes it; otherwise waits.
- **wait(chopstick[(i+1)%5]);** Tries to pick the right chopstick (using modulo for circular table).
- **eat();** Philosopher eats once both chopsticks are acquired.
- **signal(chopstick[i]);** and **signal(chopstick[(i+1)%5]); P**uts down both chopsticks, making them available for neighbors.

Conclusion: In this practical we successfully solve classical problems of synchronization using Mutex and Semaphore.

Assignment No. 4

Problem Definition:

Write a Java program (using OOP features) to implement following schedulingalgorithms:
FCFS, SJF (Preemptive), Priority (Non - Preemptive) and Round Robin (Preemptive)

1.1 Prerequisite:
Basic concepts of Scheduling, Different types of scheduling algorithms

1.2 Learning Objectives:
Understand the implementation of the Scheduling Algorithms and performance comparativestudy of Scheduling algorithms

1.3 New Concepts:

Schedular:-
Schedular is an Operating System module that selects the next job to be admitted into the system & next process to run. There are major three types of schedular basically as follows:-

1. **Short Term Schedular**
2. **Mid Term Schedular**
3. **Long Term Schedular**

Scheduling :-
Scheduling is the method specified by some means is assigned to resourcesthat complete the work; work may be either virtual computation elements like thread, processes, data flows etc. There is Major two types of scheduling algorithmto solve anysort of operations.

Preemptive Scheduling	**Non-Preemptive Scheduling**
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more compared to Non-Preemptive Scheduling.	CPU utilization is less compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.	When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.
If a high priority process frequently arrives in the ready queue, low priority	If a process with long burst time is running CPU, then another process

process may starve. with less CPU burst time may starve.
Preemptive scheduling is flexible. Non-preemptive scheduling is rigid. Ex:- Priority, Round Robin, etc. Ex:- FCFS, SJF, Priority, etc.

1.4 Scheduling Algorithms:-

1. **FCFS (First Come First Serve)**
2. **SJF (Shortest Job First)**
3. **Priority**
4. **Round Robin**

1.4.1 FCFS CPU SCHEDULING ALGORITHM :-

For FCFS scheduling algorithm, read the number of processes/jobs in the system, theirCPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each Process will be executedaccording to its arrival time. Calculate the waiting time and turnaround time of eachof the processes accordingly.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Process Wait Time : Service Time - Arrival Time P0 0 - 0 = 0
P1 5 - 1 = 4
P2 8 - 2 = 6
P3 16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

1.4.2 SJF CPU SCHEDULING ALGORITHM

For SJF scheduling algorithm, read the number of processes/jobs in the system, Their CPU burst times arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approachis to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8

Process Wait Time : Service Time - Arrival Time P0 3 - 0 = 3

P1 0 - 0 = 0

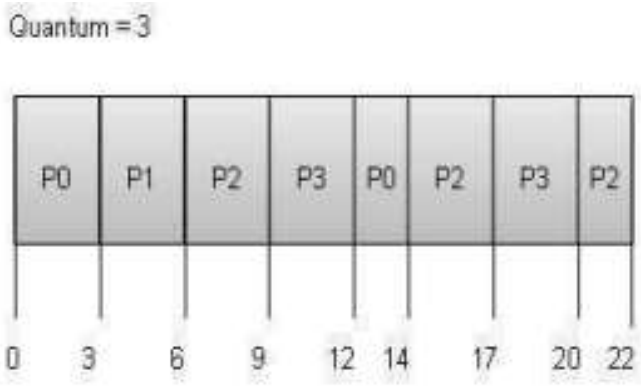
P2 16 - 2 = 14

P3 8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

1.4.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

For Round Robin scheduling algorithm, read the number of processes/jobs in thesystem, their CPU burst times, and the size of the time slice. Time slices are assignedtoeach process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time andturnaround time of each of the processes accordingly.



Process Wait Time : Service Time - Arrival Time P0 (0 - 0) + (12 - 3) = 9

P1 (3 - 1) = 2

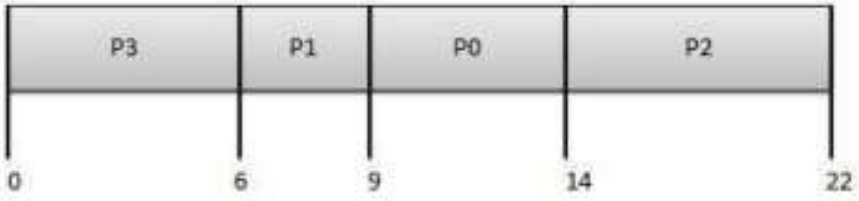
P2 (6 - 2) + (14 - 9) + (20 - 17) = 12 P3 (9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

1.2.4 PRIORITY CPU SCHEDULING ALGORITHM

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect totheir priorities. There may be two jobs in queue with the same priority, and thenFCFSapproach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Process Wait Time : Service Time - Arrival Time P0 9 - 0 = 9

P1 6 - 1 = 5

P2 14 - 2 = 12

P3 0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

Study Assignment

Assignment No 3: Write a program to recognize infix expression using LEX and YACC.

Objective: Understand the implementation of Lex and YACC Specification with simple & compound sentences.

Theory :

LEX

We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.

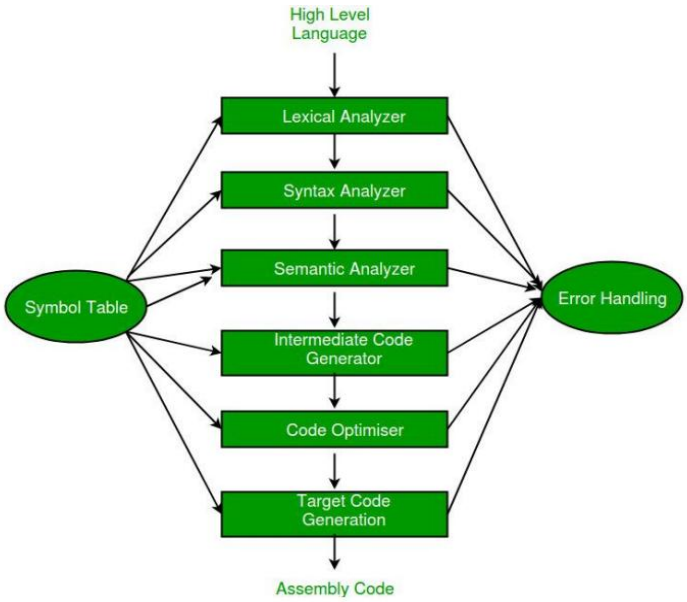


Fig:- Phases of Complier

The general format of Lex source is:

{definitions}

%%

{rules}

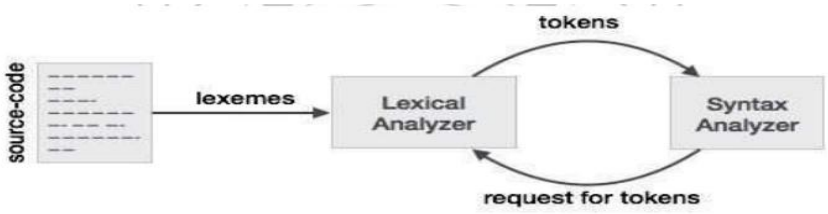
%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus %% (no definitions, no rules) which translates into a program which copies the input to the output unchanged.

Lexical Analysis:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any white space or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

Example of tokens:

▪ Type token (id, number, real, . . .)

▪ Punctuation tokens (IF, void, return, . . .)

▪ Alphabetic tokens (keywords)

Basic Functions of Lexical Analysis:

1.Tokenization i.e. Dividing the program into valid token.

2. Remove white space characters.

3. Remove comments.

4. It also provides help in generating error message by providing row number & column number.

YACC

Syntax Analyzer:

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.

Parse Tree: Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings. A parse tree is an entity which represents the structure of the derivation of a terminal string from some non-terminal (not necessarily the start symbol). The definition is as in the book. Key features to define are the root ∈ V and yield ∈ Σ* of each tree

- For each σ ∈ Σ, there is a tree with root σ and no children; its yield is σ
- For each rule A → ε, there is a tree with root A and one child ε; its yield is ε
- If t1, t2, ..., tn are parse trees with roots r1, r2, ..., rn and respective yields y1, y2, ..., yn, and A → r1r2...rn is a production, then there is a parse tree with root A whose children are t1, t2, ..., tn. Its root is A and its yield is y1y2...yn

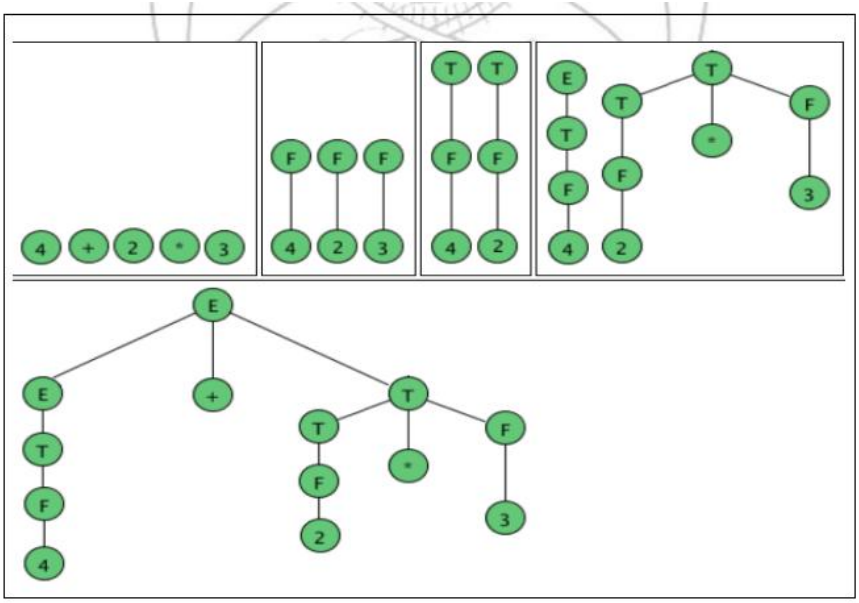
Observe that parse trees are constructed from bottom up, not top down. The actual construction of "adding children" should be made more precise, but we intuitively know what's going on. As an example, here are all the parse (sub) trees used to build the parse tree for the arithmetic expression 4 + 3 * 2 using the expression grammar

E → E + T | E - T | T

T → T * F | F

F → a | (E)

Where a represents an operand of some type, be it a number or variable. The trees are grouped by height.



Steps for Execution of Program :

1. Write lex program and save as filename. l and yacc program as filename. y.
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex filename. l to compile lex program.
4. type yacc filename. y to compile yacc program.
5. then c code will generate for both automatically.
6. type to compile both codes cc lex. yy. c y. tab. h -ll.
7. Type to run final code ./a. out.

Program

Infix.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ { yylval.dval=atoi(yytext); return NUMBER;}
[0-9]*"."[0-9]+ { yylval.dval=atof(yytext); return NUMBER;}
[a-zA-Z] { return LETTER; }
"+" { return PLUS;}
"-" { return MINUS;}
"*" { return MULTIPLY;}
"/" { return DIVIDE;}
"(" { return OPEN;}
")" { return CLOSE;}
"\n" { return ENTER;}
"$" { return 0;}
%%
```

Infix.y

```
%{
#include<stdio.h>
#include<math.h>
%}
%union {
double dval;
char symbol;
}
%token<dval>NUMBER
%token<symbol>LETTER
%token PLUS MINUS MULTIPLY DIVIDE OPEN CLOSE ENTER
%left PLUS MINUS
%left DIVIDE MULTIPLY
%nonassoc UMINUS
%type<dval>E
%%
print: E ENTER { printf("\n\v VALID INFIX EXP.....\n"); exit (0); }
;
E:E PLUS E
|
E MINUS E
|
E MULTIPLY E
```

```
|
E DIVIDE E
|
MINUS E %prec UMINUS {$$=-$2;}
|
OPEN E CLOSE { $$=$2;}
|
NUMBER { $$=$1; }
|
LETTER {$$=$1;}
;
%%

int main()
{
printf("\n Enter infix expression: ");
yyparse();
return 0;
}
void yyerror( char *msg)
{
printf("\n INVALID INFIX EXPRESSION.....: ");
}
int yywrap(){return(1);}
```

Conclusion: We learn how to recognize infix expression using LEX and YACC programs in linux. Also we learn how to recognize infix expression using LEX and YACC Programs.

Assignment No. 2: Design suitable data structures and implement Pass-I and Pass-II of a two-pass macro processor. The output of Pass-I (MNT, MDT and intermediate code file without any macro definitions) should be input for Pass-II.

Theory:

- **Macro processor.**

Macro processor is the system program which performs macro expansion i.e. replacement of macro call by corresponding set of instructions.
Therefore macro processor has to perform following main jobs:

1. Identification of macro definitions.
2. Storage of all macro definitions.
3. Identification of macro calls.
4. Replacement of macro calls by corresponding macro definition.

- **2 Pass Macro Processor**
  - PASS-1 purpose

Examines every opcode and saves all macro definition in MDT and save a copy of input text on secondary storage minus macro definition.
It also prepares MNT.

- PASS-2 purpose

Examines every opcode and replaces each macro name with appropriate macro definition.

- Specification of all the databases with their format
- **Macros with Positional and Keyword arguments**

Positional Arguments: Arguments are matched with dummy argument according to the order in which they appear.
i.e 'INCR A,B,C'
A, B & C replace the first second and third dummy argument.
Keyword argument: Allows reference to dummy argument by name as well as by position.
e.g. MAC3 &ARG1=DATA1, &ARG2=DATA2, &ARG3=DATA3

- **Conditional Macro Expansion:**

1. Conditional macros are the one, which will not get expanded to the same group of statements all the time but rather it depends on the kind of condition being put.
2. This allows conditional solution of machine instruction that appears in expansion of a macro call. The group of statements being replaced may vary in length and sequence, depending on the condition in the macro call.
3. Two macro processor pseudo opcodes, AIF & AGO are used.
4. AIF: It is a conditional branch pseudo opcode. It performs an arithmetic test and branches only if tested condition is true.
5. AGO: It is an unconditional branch pseudo-opcode or 'goto statement'.
   It specifies a label appearing on some other statement in the macro instruction definition.
6. These statements are directive to the macro processor and do not appear in macro Expansion.

Implementation Logic:

Input: A program written in assembly language containing macros with arguments and condition

Output (Expected):

The above assembly program is to be stored separately in text file. The output expected is the expanded source program, i.e. expanded macro calls.

Expected outputs are:

- The expanded source program
- Macro Definition Table (MDT).
- The Macro Name Table (MNT).
- The updated Argument List Array (ALA).

Algorithm for Pass-I:

1. /* Initialization of counters for MDT and MNT*/
   MDTC=MNTC=1;
2. Read next instruction (and divide it into its various field as label , mnemonic opcode , Arguments)
   i) /*Check for macro definition start*/
   if opcode=MACRO goto Step 5
   else /* this is not macro definition*/
   go to step 4.
   ii) (A) Write copy of instruction to output of PASS-I
   (a) Check whether opcode =END or not
   (b) If OPCODE != END goto Step 2
   (c) If OPCODE = END goto to Pass II i.e. End if this algorithm for pass I
   B) /* Start of macro definition is identified. Now Pass I will process contents of macro definition after pseudo of MACRO to MEND */
   a) Read next instruction
   (definitely this is a macro name instruction therefore as a processing of this instruction an entry will be made in MNT. ALA will be prepared for this macro, this macro name instruction will be entered in MDT*/
   b)Enter <macro-name MDTC> into MNT at MNTC
   /* current available rows in MDT and MNT are MDTC and MNTC, so macro name and its starting MDT index i.e. current value of MDTC is entered in MNT at available row i.e. MNTC*./
   c) MNTC=MNTC+1 /* To point next available row in MNT*/
   d)Prepare Argument List Array
   /* ALA is partially constructed by Pass –I to assign universal integer index to dummy arguments*/
   e) Enter macroname instruction in MDT at MDTC.
   f) MDTC=MDTC+1.
3. /* Process other instruction in macro definition inducing MEND instruction*/
   a. Read next card
   b. Substitute Index notations for dummy arguments.
   c. Enter this instruction (where dummy arguments are replaced by integer indices) into MDT.
   d. MDTC:= MDTC +1
   e. If OPCODE of this instruction is MEND then goto Step 2. else goto Step 6.a.

Algorithm for PASS II:

1. Read next instruction from source program outted by Pass-I and divide it into fields as label, mnemonic opcode, arguments.
2. Search through MNT to find match for OPCODE of instruction read in Step1 with macronames in MNT.
3. /* If no Macro call found*/

   if no match then it indicates that this instruction is not a macro call instruction and hence

   (a) Write this instruction to expanded source program file
   (b) Check whether OPCODE of this instruction is END or not.
   (c) If not END then it indicates that this is not end of source program and hence go to step 1
   (d) If OPCODE=END then this indicates end of source program and hence give the output of Pass II i.e. Expanded source program to assembler (This is the end of Pass II of Macroprocessor)

4. /* if macro name found */

   If OPCODE of (instruction read in step 1)= any macro name of MNT then it indicate that this instruction is macrocall instruction and hence.

   a. Obtain corresponding MDT index and assign to MDTP.
   b. Set up ALA (for the association of Integer Indices and actual parameters)
   c. MDTP:=MDTP+1
   d. Get next instruction from MDT
   e. Substitute actual arguments instead of integer indices.
   f. If OPCODE of this instruction is not MEND then write this instruction (after replacing integer indices by actual arguments) to the expanded source program.
   g. If OPCODE of this instruction is MEND then goto Step 1 again.

In Brief steps:

- Recognizing the Macro Definition.
- Save the Macro name in the MNT, and the Definition in the MDT.
- Recognize the Macro call and
- Use of ALA
- Expand the macro call.

Conclusion: Hence we successfully designed suitable data structures and implemented Pass-I Pass-II of a Two-pass macroprocessor.

Assignment No. 1: Design suitable Data structures and implement Pass-I and Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. The output of Pass-I (intermediate code file and symbol table) should be input for Pass-II.

Theory:

Function of the assembler.

An assembler accepts as input an assembly language program and produces as output its machine language equivalent.

Two Pass assembly: -

1. It can handle forward references easily.
2. Location counter (LC) processing is done in pass 1 and symbols defined in the program are entered in the symbol table.
3. The second pass uses this address information to generate target program for the loader.
4. In this scheme pass-1 constructs an intermediate representation of source program used by pass-2. This is called intermediate code (IC).

Single pass assembly: -

1. Here problem of forward reference is resolved using the technique of back-patching where the operand that is forward referenced is left blank initially and then is later filled.
2. These blank field are kept in a special table called as table of incomplete information. (TII)
3. Each entry in TII is of the form: (<Instruction address> <symbol>)
4. e.g. we have forward reference in following statement:

```
- - - -
MOVER, B, ONE
- - - - - .
- - - - - .
ONE DC 1
- - - - - .
```

This statement can be partially synthesized since ONE is a forward reference.
Hence entry in TII will be of the form (1001, ONE)

5. By the time we process END statement. TII will contain information about all forward references.
6. Assembler can process this entry by checking out the symbol table and finding the address of each forward reference.

Tasks Performed by the 2 PASS Assembler:

Pass I:

1. Separate symbol table mnemonic and operand fields
2. Build the symbol table.
3. Perform IC processing.
4. Construct intermediate code.

Pass II:

1. Synthesis of target program.
2. Evaluate fields and generate code.
3. Process pseudo –opcodes.

Input :

A sample program in assembly language

A machine operation table (MOT) specifying the Mnemonics op- code, machine opcode, length of instruction op- code, instruction length, instruction format etc.

A pseudo operation table (POT) specifying the Pseudo-ops.

Output (Expected):

The above assembly program should be stored in a text file which would be the source file for the assembler.

The output of the assembler is expected as follows:

After Pass I: construction of symbol table and literal table, pool table

(Draw tabular structure of symbol table and literal table, pool table**)**

Algorithm: For PASS I of TWO PASS ASSEMBLER

1. Start
2. Set initial values
LC=0, PoolTabptr=0, LTABptr = 1,
POOLTAB[1] =1
Symbol_No=1
3. /* Scan and process each statement till end statement encounters*/
a. If label is present then
i) This_label=symbol in label field
ii) Enter (this_label, LC) into SYMBOL TABLE.

b. If START statement then i.e. if opcode='START' then
i) Generate intermediate code(AD,01)
ii) If argument is provided to START statement then Reset value of LC = argument given on START statement. Change address of label (if any) on START start statement to value of LC obtained in above step (This address is reflected in symbol table entry for this label.)

c. If ORIGIN statement i.e. if OPCODE = 'ORIGIN' then
i) Calculate address specified specified in operand field of ORIGIN statement
ii) Reset LC=address calculated in above step.
(No intermediate code is generated for ORIGIN statement)

d. If EQU statement i.e. if OPCODE='EQU' then
i) Calculate address specified in operand field of 'EQU' statement.
ii) Correct symbol table entry for this label by changing its address to the address calculated in above step
(No intermediate code is generated for EQU statement)

e. /* Process declarative statement
i) If OPCODE='DC' then
(1) Generate code (AD,05)
(2) Size =size of constant
(3) LC=LC+size
ii) If OPCODE='DS' then

(1) Generate code(AD,06)
(2) Size=size specified as argument of DS statement also
reflect this size in symbol table.
(3) LC=LC+size

f. /*Process LTORG statement
If LTORG statement i.e. if OPCODE='LTORG' then
(1) Process LITTAB[POOLTAB[PTP]…] to LITTAB[LTP-1] So that memory location can be allocated to these literals and put addresses in address field of literal Table. Also update LC accordingly.

(2) PTP=PTP+1
(3) POOLTAB[PTP]=LTP

g. /*Process Imperative statement*/
If IMPERATIVE statement then
(1) Code1:=Machine code for given OPCODE in MOT
(2) LC:=LC+size of instruction (obtained from MOT)
(3) Code:=(IS,code1)

4. /* Process operands which can be registers, symbol i.e.(storage variables or constant) or integer constant.
If OPERAND is a register then Register operand can be either of AREG, BREG, CREG, DREG therefore corresponding code generated is 1,2,3,4 resp.
If OPERAND is symbol then Entry-no=symbol table entry number of this symbol Generated code as (S,Entry-No)
If OPERAND is a literal This-literal:= literal in operand field Generate code as (L, this-literal no in literal table) LTP:=LTP+1
If OPERAND is integer constant then This-constant=constant value Generate code(C, this-constant)

5. /*Process END statement*/
a. If END statement i.e. if OPCODE='END' statement then Perform 3.f
b. Generate (AD,02)
c. GOTO Pass II

Algorithm: For PASS II of TWO PASS ASSEMBLER

1. Start
2. Initialize various pointers
PTP=1
LC:=0
3. /* Read intermediate code i.e. set of intermediate instructions and depending upon intermediate opcode and intermediate operand code process it as follow till intermediate code for END statement encounters*/
(a) Clear machine-code-buffer
(b) /* START/ORIGIN statement*/
if START/ORIGIN statement i.e. if intermediate code for START/ORIGIN statement is encountered then
    a. LC:= value specified in it's operand field
    b. Size:=0 (as no machine code is generated for START /ORIGIN construction)
(c) /*Process DC statement*/
if DC statement i.e. if intermediate code for DC statement is encountered then
i. Machine opcode field will have blank or zero
ii. First operand field will have also blank or zero
iii. Second operand field will contain value of constant being defined (This is how constant being declared is stored at memory location pointed by current value of LC)
iv. This code is assembled in machine-code buffer
v. Size:=size of DC statement (which can be obtained from POT
vi. LC=LC+size

(d) /*Process DS statement*/
if DS statement i.e. if intermediate code for DS statement is encountered then
    a. size:= argument value specified in DS statement
    b. memory locations of 'size' number of times are reserved
    c. and all three fields of those memory locations are left blank

d. LC:=LC+size

i. /*Process LTORG statement*/

If LTORG statement is encountered i.e. If intermediate code for LTORG is encountered then

(i) No-Of-literals-in –this- POOL=0 Process literals LITTAB[POOLTAB[PTP] to LITTAB[PTP}

(ii) Process literals in current pool in the same as processing of constants in DC statements

(A) process each literal in current pool, so that literals will be defined at the memory locations pointed by LC.

(1) M/C opcode field will have blank/zero.

(2) OPERAND1 field will have blank/zero.

(3) OPERAND1 field will have literal-value.

(4) LC:=LC+1

(5) No-of-Literals-in this Pool=No-of literals-in this Pool+1

(Repeat steps 1-5 for each literal)

(B) Size:=no-of literals-in this-Pool.

(C) PTP:=PTP+1

ii. /*Process imperative statements*/

if IMPERATIVE statement is encountered i.e. Intermediate code for imperative statement is encountered then

(i) Machine code field will have only equivalent machine code for imperative opcode. For example, if intermediate opcode is (IS, 01) then discard IS and store only 01 in machine opcode field.

(ii) OPERAND1 will have machine constant representing machine register.

(iii) By referring second operand field of intermediate instruction Pass-II can decide whether second operand is symbol or LITERAL.

(iv) Depending upon type of second operand obtain its address by referring to SYMBOL TABLE or LITERAL TABLE.

(v) OPERAND2 field of machine code will contain address obtained in above step.

(vi) Size:=size of instruction

iii. IF size#0 then

i. Move contents of machine code buffer to address=code area addr + LC

ii. LC: = LC + size.

4. /*Process END statement*/

If END statement is encountered i.e. intermediate code for END statement is encountered then

(i) Perform STEP (2.f) and (2.b). Because processing of LTORG and END statement is same as both of them cause the allocation of addresses to the literals in current-pool. (i.e. literal pool ending at LTORG or END statement)

(ii) AS it is end of program contents of code area are moved into OUTPUT file.

Conclusion: Hence we have successfully designed suitable data structures and implemented Pass I and Pass II of a two pass assembler for pseudo-machine and generated intermediate code file and symbol table for Pass I and machine code for Pass II.