

EXPERIMENT NO: -01 (Group A)**DATE:**

1.1 Title: 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 file and symbol table) should be input for pass-II.

1.2 Aim: To implement pass-I of a two-pass assembler for pseudo machine in Java.
To implement Pass-II of two pass assembler for pseudo-machine in Java.

1.3 Objectives:

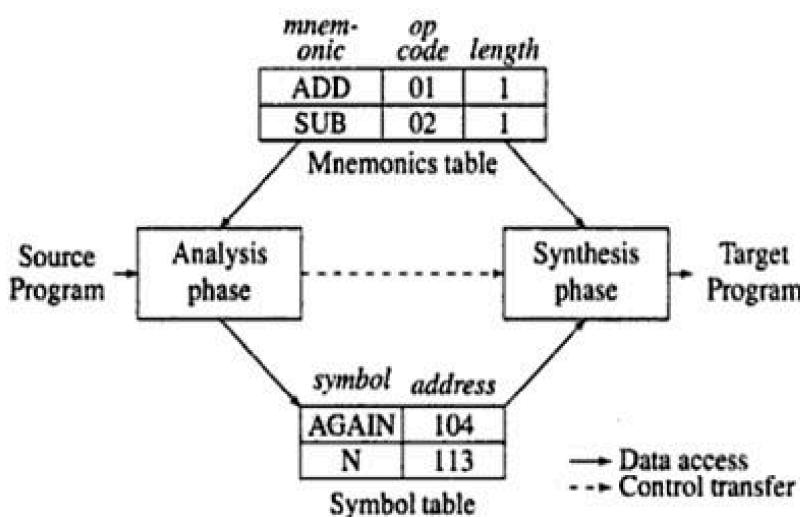
- To understand working of pass I of 2 pass assembler
- To know data structures required
- To apply OOP concepts of Java for implementation of assembler
- To understand working of pass II of 2 pass assembler
- To know data structures required
- To apply OOP concepts of Java for implementation of assembler

1.4 Hardware used: Experimental Setup/Instruments/ Devices**1.5 Software used (if applicable) / Programming Languages Used:****1.6 Theory:-**

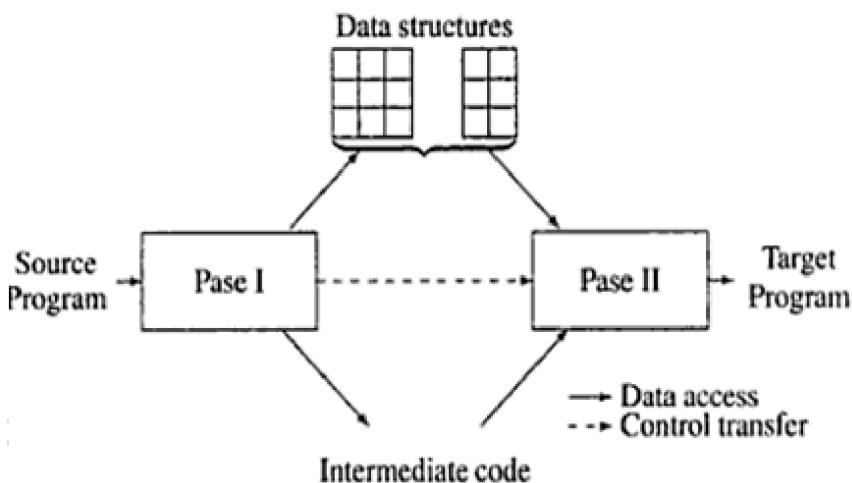
An assembler is a program that accepts an assembly language program as input and produces its output as machine code. An assembler has to perform certain functions for translating a program from assembly language to machine language.

They are:-

- 1) Replace symbolic address by numeric ones.
- 2) Replace symbolic equation codes by machine operation codes.
- 3) Reserve storage and instruction for data.

**Data Structures**

- 1) SYMTAB: Stores the symbols and their address.
- 2) LITTAB: Stores the literals and their address.
- 3) POOLTAB: Stores the starting of various literals.
- 4) OPTAB: Stores the machine code for mnemonics.



Data structures

SYMTAB, LITTAB, POOLTAB produced by pass-I

1.7 Algorithm:

Input—OPTAB, Source program

Output—SYMTAB, LITTAB, POOLTAB, IC

Steps:

1) Initialize all pointers.

`loc_cntr := 0 ; (Default value)`

`pool_tab[1] := 1 ;`

`pooltab_ptr := 1 ;`

`littab_ptr := 1 ;`

2) While next statement is not an END statement

a) If label is present then

`this_label := symbol in label field ;`

Enter(`this_label, loc_cntr`) is SYMTAB.

b) If a LTORG statement then

Process literals LITTAB [POOLTAB[pooltab_ptr]].. LITTAB[littab_ptr-1] to allocate memory and put the address field. Update `loc_cntr` accordingly

i. `Pooltab_ptr := Pooltab_ptr + 1`

ii. `POOLTAB [Pooltab_ptr] := littab`

c) If a START or ORIGIN statement then

`loc_cntr := value specified in operand field`

d) If an EQU statement then

i. `this_addr := value of <address spec> ;`

ii. Correct the symtab entry for `this_label` to (`this_label, this_addr`);

e) If a declaration statement then

i. `Code := code of declaration statement ;`

ii. `size := size of memory area required by DC/DS`

iii. `loc_cntr := loc_cntr + size ;`

iv. generate IC '(DL, code)...

f) If an imperative statement then

i. `Code := machine opcode from OPTAB`

ii. Loc_cntr := loc_cntr + instruction length from OPTAB;
 iii. If operand is a literal then
 this_literal := literal in operand field;
 LITTAB [littab_ptr] := this_literal;
 littab_ptr := littab_ptr + 1;
 else (i.e., operand is a symbol)
 this_entry := SYMTAB entry number of operand;
 Generally IC ‘(IS , Code) (S, this_entry)’ ;

3). (Processing of END statement)

- a) Perform step 2(b)
- b) Generate IC ‘(AD, 02)’.
- c) Go to pass-II

Algorithm

Input- SYMTAB, LITTAB, POOLTAB, IC produced by pass-I
 Output-Target program in machine language.

Steps:

1. Code_area_address = address of code area
 pooltab_ptr = 1
 loc_cntr = 0
2. While next statement is not END statement
 - a) clear machine_code_buffer
 - b) If an LTORG statement then
 - a) Process literals
 - b) size = size of memory area required for literal
 - c) pootab_ptr = pootab_ptr ++
 - c) If START or ORIGIN statement then
 - a) loc_cntr = value specified in operand field
 - b) size = 0
 - d) If a declaration statement then
 - a) If a DC statement then

Assemble the constant in machine_code_buffer
 - b) Size = size of memory area required by DC/DS
 - e) If an imperative statement
 - a) Get operand addr from SYMTAB or LITTAB
 - b) Assemble instruction in machine_code_buffer
 - c) Size = size of instruction
 - f) If size is not equal to 0
 - a) move contents of machine_code_buffer to the address
 $code_area_address + loc_cntr$
 - b) loc_cntr = loc_cntr + size
3. (Processing of END statement)
 - a) Perform step 2(b) and 2(f)
 - b) Write code area into output file

1.8 Example:

Mnemonic operation code:-

Symbolic Opcode (Mnemonic)	Machine Code for Opcode	Size of Instruction	Type
STOP	00	1	IS
ADD	01	1	IS
SUB	02	1	IS
MULT	03	1	IS
MOVR	04	1	IS
MOV M	05	1	IS
COMP	06	1	IS
BC	07	1	IS
DIV	08	1	IS
READ	09	1	IS
PRINT	10	1	IS
START	01	-	AD
END	02	-	AD
ORIGIN	03	-	AD
EQU	04	-	AD
LTORG	05	-	AD
DS	01	-	DL
DC	02	1	DL

Example:-**Source Program**

START
 READ N
 MOVER B,=’1’
 MOVM B,TERM
 AGAIN: MULT B,TERM
 MOVER C,TERM
 COMP C,N
 BC LE,AGAIN
 MOVEM B,RESULT
 LTORG
 PRINT RESULT
 STOP
 N DS 1
 RESULT DS 20
 TERM DS 1
 END

LC	Intermediate Code
1000	(AD,01)(C,1000)
1000	(IS,09)(S,0)
1001	(IS,04)(RG,02)(L,0)
1002	(IS,05)(RG,02)(S,1)
1003	(IS,03)(RG,02)(L,0)
1004	(IS,04)(RG,03)(S,1)
1005	(IS,06)(RG,03)(S,0)
1006	(IS,07)(CC,04)(S,2)
1007	(IS,05)(RG,02)(S,3)
1008	(DL,02)(C,01)
1009	(IS,10)(S,3)
1010	(IS,00)
1011	(DL,01)(C,01)
1012	(DL,01)(C,20)
1032	(DL,01)(C,01) (AD,02)

OUTPUT OF PASS-I (INPUT FOR PASS-II):-

SYMBOL TABLE		LITERAL TABLE		POOL TABLE	
	SYMBOL	LITERAL	ADDRESS		0
0	N	='1'	1008		
1	TERM				
2	AGAIN				
3	RESULT				

Output of Pass II:

Intermediate Code

- (AD,01)(C,1000)
- (IS,09)(S,0)
- (IS,04)(RG,02)(L,0)
- (IS,05)(RG,02)(S,1)
- (IS,03)(RG,02)(L,0)
- (IS,04)(RG,03)(S,1)
- (IS,06)(RG,03)(S,0)
- (IS,07)(,CC,04)(S,2)
- (IS,05)(RG,02)(S,3)
- (DL,02)(C,01)
- (IS,10)(S,3)
- (IS,00)
- (DL,01)(C,01)
- (DL,01)(C,20)
- (DL,01)(C,01)
- (AD,02)

LC	Machine Code
1000	09 00 1011
1001	04 02 1008
1002	05 02 1032
1003	03 02 1032
1004	04 03 1032
1005	06 03 1011
1006	07 04 1003
1007	05 02 1012
1008	00 00 0001
1009	10 00 1012
1010	00 00 0000
1011	
1012	
1032	

1.9 OOP concepts used:

1.10 Conclusion:

In this way, we have studied that Pass I accepts MOT, POT, and source program as input and generates SYMTAB, LITTAB and intermediate code as output for pass II. Also we have studied that pass II accepts SYMTAB, LIOTTAB and intermediate code from pass I and generates target program.

1.11 Questions

- Q.1 What is assembler?
- Q.2 Which data structures are used by Pass I of 2 pass assembler?
- Q.3 What are 2 variants of intermediate code?
- Q.4 What are the characteristics of an intermediate code?
- Q.5 Why there is need of 2 pass assembler?
- Q.6 What is forward reference?
- Q.7 How one pass assembler handles forward reference?

Q.8 Which data structures are used by pass II of 2 pass assembler?

Signature of Staff with date

M.V.P.S.K.B.T.C.O.E.

EXPERIMENT NO: -02 (Group A)**DATE:-**

1.1 Title: - Design suitable data structures and implement pass-I and pass-II of a two pass macro processor. The output of pass-I(MNT, MDT, intermediate code file without any macro definitions)should be input for pass-II.

1.2 Aim: To implement Pass-I of two pass macro-processor for pseudo-machine in Java.
To implement Pass-II of two pass macro-processor for pseudo-machine in Java

1.3 Objectives:

- To know macro
- To understand working of pass I and pass II of 2 pass macro-processor
- To know data structures required
- To apply OOP concepts of Java for implementation of macro-processor

1.4 Hardware used: Experimental Setup/Instruments/ Devices:**1.5 Software used (if applicable) / Programming Languages Used:****1.6 Theory:-****Macro definition**

Macro allows a sequence of source language code to be defined once and then referred to by name each time it is to be referred. Each time this name occurs in a program, the sequence of codes is substituted at that point.

A macro consists of:

- (1) Name of the macro
- (2) Set of parameters

Body of macro (definition) Parameters in a macro are optional

For example,

```
MACRO  INCR
&AREG
ADD AREG, &ARG
ADD BREG, &ARG
ADD CREG, &ARG
MEND
```

Macro preprocessor

Macro preprocessor takes a source program containing macro definitions and macro calls and translates into an assembly language program without any macro definitions or calls .This program can now be handed over to a conventional assembler to obtain the target language. It is possible to implement a macro pre-processor which processes macro definitions and macro calls for the purpose of expansions.

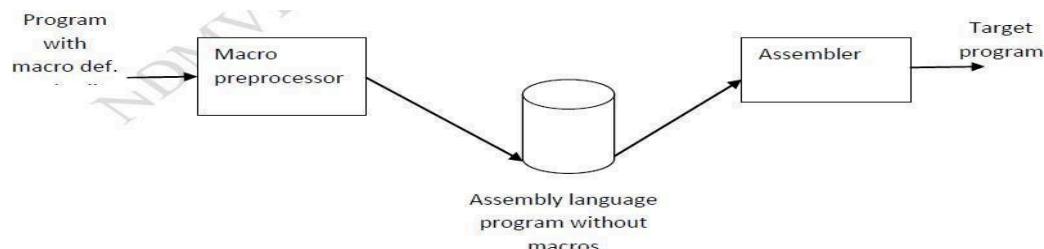


Fig. A scheme for a macro pre-processor

Features of macros:-

- Macro with arguments
- Formal parameters
- Actual parameters
- Positional parameters
- Keyword parameters
- Macro with default arguments
- Conditional macro expansion
- Macro processor pseudo-ops AIF and AGO
- Macro call within macro
- Macro definition within macro

Implementation:-

Function	Pass
1. Recognize macro definitions 2. Save the definitions	Pass I
3. Recognize macro calls 4. Expand calls and substitute arguments	Pass II

Databases used in Pass-I of two pass macro preprocessor:

1. The input macro source deck.
2. The output macro source deck copy for use by pass2
3. The Macro Definition Table (MDT), used to store the body of the macro definitions.
4. The Macro Name Table (MNT), used to store the name of the defined macros
5. The Macro Definition Table counter (MDTC), used to indicate the next available entry in the MDT.
6. The Macro Name Table counter (MNTC), used to indicate the next available entry in the MDT.
7. The argument list array (ALA), used to substitute index markers for dummy argument before storing a macro definition.

Databases used in Pass-II of two pass macro preprocessor:

1. The copy of the input macro source deck.
2. The output expanded macro source deck to be used as i/p to the assembler.
3. The Macro Definition Table (MDT), created by pass I.
4. The Macro Name Table (MNT), created by pass I.
5. The Macro Definition Table pointer (MDTP), used to indicate the next Line of text to be used during macro expansion.
6. The argument list array (ALA), used to substitute macro call arguments for the Index markers in the stored macro definition.

An Example: (input of pass-1)

```

START 100
MACRO
INCR &X,&Y,&REG=AREG
MOVER &REG,&X
ADD &REG,&X MOVEM
&REG,&X MEND
MACRO
DECR &A,&B,&REG=BREG

```

```

MOVER &REG,&A
SUB &REG,&B
MOVEM &REG,&A
MEND
READ N1
READ N2
INCR N1, N2, REG=CREG
DECR N1, N2
STOP
N1 DS 1
N2 DS 1
END

```

Intermediate code (output of pass-1):-

```

START 100 READ
N1 READ N2
INCR N1,N2,REG=CREG DECR
N1,N2
STOP
N1 DS 1
N2 DS 1
END

```

MDT:

```

1      INCR &X,&Y,&REG=AREG 2
2      MOVER #3,#1
3      ADD #3,#2
4      MOVEM #3,#1
5      MEND
6      DECR &A,&B,&REG=BREG 7
7      MOVER #3,#1
8      SUB #3,#2
9      MOVEM #3,#1
10     MEND

```

MNT:-

Sr.No.	Name	MDT Index
1	INCR	1
2	DECR	6

MDTC=11

MNTC=3

- #1: First parameter
- #2: Second parameter
- #3: Third parameter

Pass-II will create argument list array and expand the macro.

01) Macro call

INCR N1,N2,REG=CREG ALA:-

1 N1
2 N2
3 CREG

Expanded code:-

MOVER CREG, N1
ADD CREG, N2
MOVEM CREG, N1

02) Macro call

DECR N1,N2

ALA:-

1 N1
2 N2
3 AREG
Default value

Expanded code:-

MOVER AREG,N1
ADD AREG, N2
MOVEM AREG, N1

Intermediate code (i/p for pass-2):-

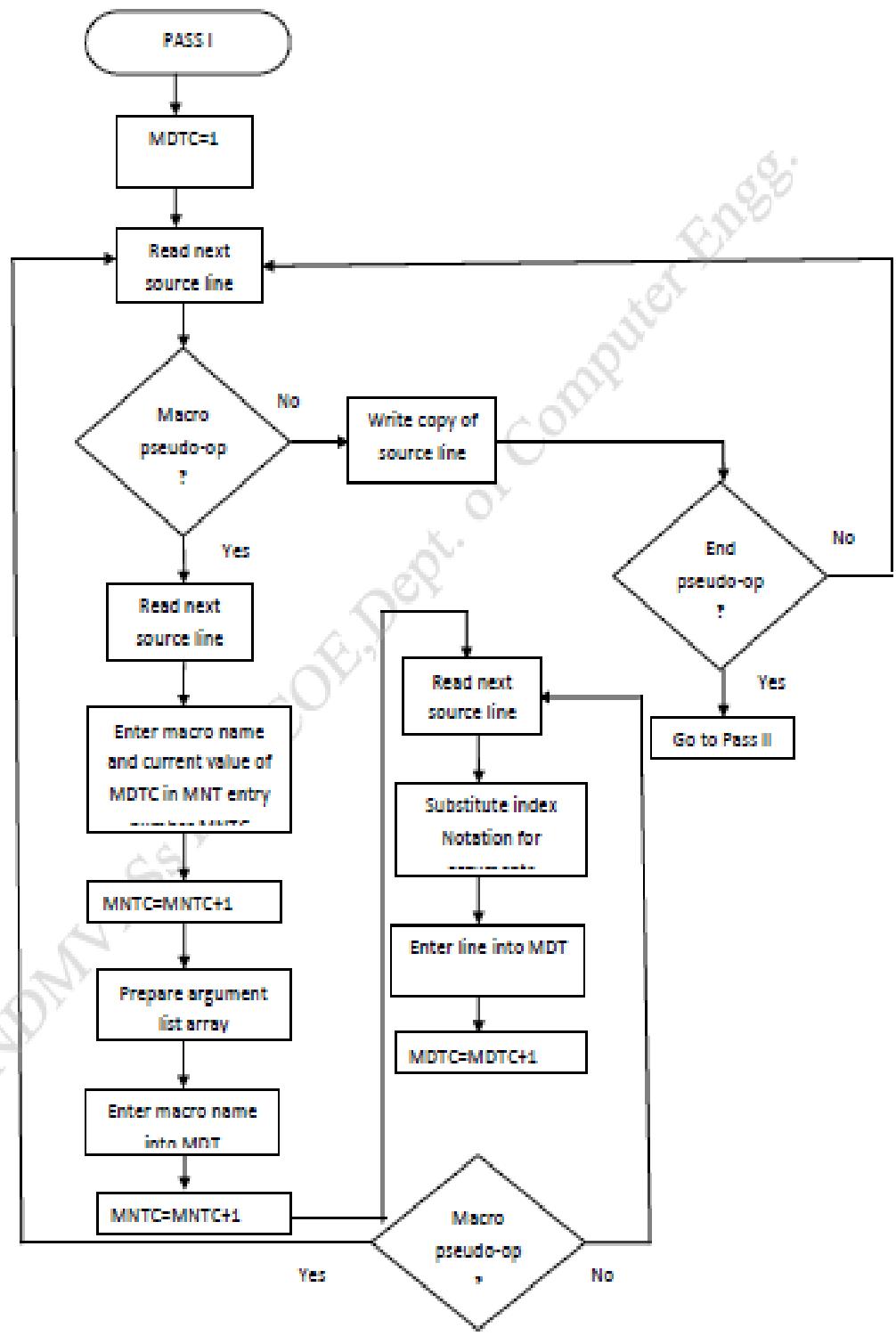
START 100
READ N1 READ
N2
INCR N1,N2,REG=CREG
DECR N1,N2
STOP
N1 DS 1
N2 DS 1
END

Intermediate code(O/P for pass-2):

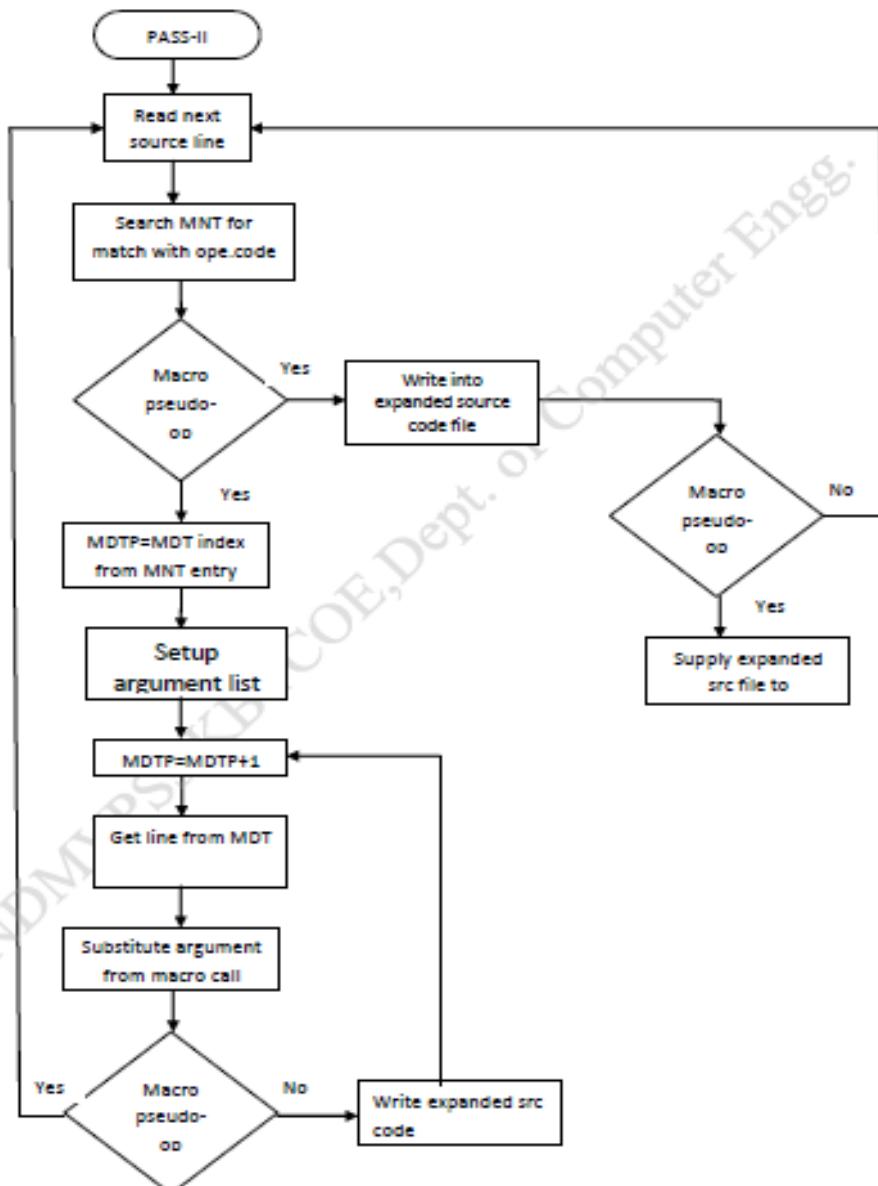
START 100
READ N1 READ
N2
INCR N1,N2,REG=CREG
MOVER CREG, N1
ADD CREG,N2 MOVEM
CREG, N1 MOVER
AREG,N1 ADD AREG,
N2 MOVEM AREG, N1
STOP
N1 DS 1
N2 DS 2
END

1.7 Algorithm for Pass I:-

M.V.P.S.K.B.T.C.O.E.



Algorithm for Pass II



OOP concepts used:

Conclusion

In this way we have studied that recognize macro definition and store macro definitions done in pass I of 2 pass macro processor and it also constructs MDT and MNT data structures and also recognized macro call and expansion of macro call is done in pass II of 2 pass macro processor using MDT and MNT data structures constructed by pass I.

1.8 Questions:

1. What is macro?
2. What is difference in macro and function?
3. What is function of macro processor?
4. Which data structures are used by pass I of 2 pass macro processor?
5. Which data structures are used by pass II of 2 pass macro processor?
6. What is macro call within macro? How it is processed?

Signature of Staff with date

M.V.P.S.K.B.T.C.O.E.

EXPERIMENT NO: -03 (Group A)

DATE:-

1.1 Title: - Write a program to create a Dynamic Link Library for any mathematical operation and write a application program to test it. (Java Native Interface/Use VB or VC++)

1.2 Aim: To write an application program to test Dynamic Link Library for mathematical operations.

1.3 Objectives: - To design Dynamic Link Library

- To use Dynamic Link Library in an application program

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Introduction

In a conventional non-shared, "static" library, sections of code are simply added to the calling program when its executable is built at the "linking" phase; if two programs call the same routine, the routine is included in both the programs during the linking stage of the two. With dynamic linking, shared code is placed into a single, separate file. The programs that call this file are connected to it at run time or at the time of binding. In dynamic linking, the required functions are compiled and stored in a library with extension .DLL. Unlike static linking, no object code is copied in to the executable from the libraries. Instead of that, the executable will keep the name of the DLL in which the required function resides and when the executable is running, it will load the DLL and call the required functions from it. These methods yield an executable with small footprint but have to compromise on speed a little.

Feature of DLL

The code in a DLL is usually shared among all the processes that use the DLL; that is, they occupy a single place in physical memory, and do not take up space in the page file.

Steps to create DLL in Visual Basic:

1. Create a New project => select **Class Library**. Give it a name of MyLib. Click OK.
2. Write code for your functions: For example, to program a simple Math function, which will use 2 textboxes from your application, add 2 numbers together and then display the result in a label.

Example1:

```
Public Class MyFunctions
    Public Function AddMyValues(ByVal Value1 As Double, ByVal Value2 As
        Double) Dim Result As Double
```

```
        Result = Value1 + Value2
```

```
        Return Result
```

```
    End Function
```

```
    End Class
```

3. Save the project, and then build it.

4. Create a New project => select **Windows Forms Application**. Add 2 textboxes for input, label **lblResult** for result and the button **btnAdd**.

5. Add a reference to your newly created DLL as Project -> Add Reference, and browse to where your DLL MyLib, select it and click ok.
6. Add **Imports** statement for DLL.
7. Now in the Button_Click event, call the desired function from DLL.

Imports MyLib Public Class Form1

```
Private Sub btnAdd_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles btnAdd.Click  
    Dim Add As New MyLib.MyFunctions
```

```
lblResult.Text = Add.AddMyValues(CDbl(TextBox1.Text), CDbl(TextBox2.Text)).ToString
```

```
End Sub
```

```
End Class
```

8. Save the project and then run the application. Enter 2 numbers and then press the button and the result - which was done in the DLL will be displayed in your result label.

1.7 Conclusion: Thus we can create DLL for any functionality which is required frequently and use it whenever required.

1.8 Questions:

Q.1 What is DLL?

Q.2 Compare DLL with function.

Signature of Staff with date

1.1 Title: Write a program to solve Classical Problems of Synchronization using Mutex and Semaphore.

1.2 Aim: To implement classical problem of synchronization using mutex and semaphore.

1.3 Objectives:

- To understand reader writer synchronization problem
- To solve reader-writer synchronization problem using mutex and semaphore

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1. Java
2. Eclipse/NetBeans

1.6 Theory:-

- There is a data area shared among a number of processor registers.
- The data area could be a file, a block of main memory, or even a bank of processor registers.
- There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).
- The conditions that must be satisfied are
 - Any number of readers may read simultaneously read the file.
 - Only one write at a time may write to the file.
 - If a writer is writing to the file, no reader may read it.

Semaphore:

Definition: Semaphores are system variables used for synchronization of process

Two types of Semaphore:

□ **Counting semaphore** – integer value can range over an unrestricted domain

□ **Binary semaphore** –

Integer value can range only between 0 and 1; can be simpler to implement · Also known as mutex locks

Semaphore functions:

Package: import java.util.concurrent.Semaphore;

1) **To initialize a semaphore:**

```
Semaphore Sem1 = new Semaphore(1);
```

2) **To wait on a semaphore:**

```
/* Wait (S)
   while S<=0
   no-op;
   S--;
*/
```

```
Sem1.acquire();
```

3) **To signal on a semaphore:**

```
/* Signal(S)
   S++;
*/
```

```
mutex.release();
```

Algorithm/Flowchart:

Algorithm for Reader Writer:

1. import java.util.concurrent.Semaphore;
2. Create a class RW
3. Declare semaphores – mutex and wrt
4. Declare integer variable readcount = 0
5. Create a nested class Reader implements Runnable
 - a. Override run method (Reader Logic)
 - i. wait(mutex);
 - ii. readcount := readcount +1;
 - iii. if readcount = 1 then
 - iv. wait(wrt);
 - v. signal(mutex);
 - vi. ...
 - vii. reading is performed
 - viii. ...
 - ix. wait(mutex);
 - x. readcount := readcount – 1;
 - xi. if readcount = 0 then signal(wrt);
 - xii. signal(mutex);
6. Create a nested class Writer implements Runnable
 - a. Override run method (Writer Logic)
 - i. wait(wrt);
 - ii. ...
 - iii. writing is performed
 - iv. ...
 - v. signal(wrt);
7. Create a class main

a. Create Threads for Reader and Writer**b. Start these thread****Input:**

1. Number of Readers
2. Number of Writers

Output:

1. Execution of Readers and Writers

1.7 Conclusion: : Thus we have implemented Reader Writer synchronization problem using semaphores in Java

1.8 Questions

- Q.1 What is synchronization of threads?
- Q. 2 Explain reader writer problem
- Q.3. Explain wait and sequence functions
- Q. 4. What is semaphore.
- Q. 5. What are different types of semaphore

Signature of Staff with date

1.1 Title: Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF(Preemptive), Priority (Non-preemptive) and Round Robin (Preemptive)

1.2 Aim: To implement program to simulate CPU Scheduling algorithms: FCFS, SJF(Preemptive), Priority (Non-preemptive) and Round Robin (Preemptive)

1.3 Objectives: - To study and implement scheduling algorithms

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

CPU scheduling program leads with a program of deciding which of the process in ready queue to be allocated the CPU.

Performance criteria that are frequently used by scheduler to maximize system performance are:

- i. **CPU utilization:** The idea is to keep the CPU busy 100 percent of time.
 - ii. **Throughput:** It refers to no. of processes executed per unit time.
 - iii. **Turnaround time:** It is defined as interval from the time of submission of a process to the time of its completion.
 - iv. **Waiting Time:** waiting time = turnaround time - processing time.
 - v. **Response Time:** The time from the submission of request until the first response is produced. Scheduling algorithm can be classified into two categories:
- a) Preemptive scheduling.
 - b) Non-preemptive scheduling.

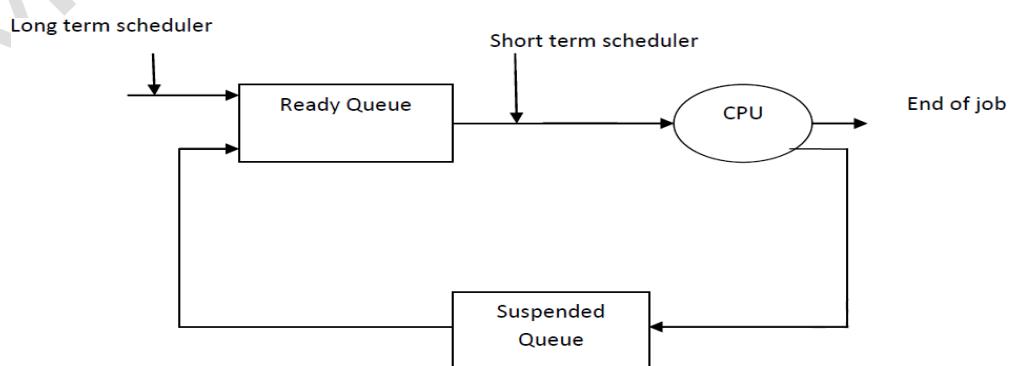
- a) **Preemptive scheduling:** In preemptive scheduling, the CPU can be taken away from the process. A process can be temporarily suspended due to:
- i) Request for I/O
 - ii) Time slab over

- b) **Non-preemptive scheduling:** In Non-preemptive scheduling, a process runs to its completion. Once a process has been given the CPU, the CPU cannot be taken away from that process.

Types of Scheduler:

1. Long term scheduler:

Long term scheduler decides which job shall be admitted in the ready queue for processing.



2. Short term scheduler:

It allocates the CPU to processes belonging to ready queue for immediate processing. It is often called at

- i) End of time slab.
- ii) In case of I/O request.

3. Medium term Scheduler:

Sometimes it is necessary to remove processes from main memory to hard disk to create more memory for other processes. At some later time they can again be reloaded into memory. The suspended processes are swapped out and in by medium term scheduler.

Some common scheduling algorithms are:

- i) First-Come-First-Served (FCFS) scheduling.
- ii) Shortest-Job-First (SJF) scheduling.
- iii) Round Robin (RR) scheduling.
- iv) Priority Scheduling.

1) FIRST COME FIRST SERVE (FCFS):

FCFS is the simplest scheduling algorithm. Implementation of the FCFS is easily managed with a FIFO queue. New process enters the ready queue at the front and the running process is removed from the rear of the queue.

Algorithm:

1. Read no. of processes N.
2. Read burst time (BT) for each process and enter it in the queue.
3. Start time (ST) = 0.
4. While queue not empty
 - a. Remove process from the queue.
 - b. Use ST and BT to calculate Finish Time (FT) of process.
 - c. Use ST and FT to calculate waiting time (WT) and turnaround time (TT) for process.
5. Stop

2) SHORTEST JOB FIRST (SJF):

Always the process having min. burst time is allocated CPU.

Algorithm:

1. Read no. of processes N.
2. Read burst time (BT) for each process and enter it in the list.
3. Start time (ST) = 0.
4. While queue not empty
 - a. Remove process with min. BT from the list.
 - b. Use ST and BT to calculate Finish Time (FT) of process.
 - c. Use ST and FT to calculate waiting time (WT) and turnaround time (TT) for process.
5. Stop

3) ROUND ROBIN (RR):

Processes are allocated CPU for amount of time equal to Time Quantum (TQ). If process finishes, it releases CPU. If it does not finish, then CPU is preempted from the process and given to next process in the queue.

Algorithm:

1. Read no. of processes N.
2. Read burst time (BT) for each process and enter it in the queue.
3. Read Time Quantum (TQ) from the user.
4. Start time (ST) = 0.
5. Initialize allocation time (AT) for each process to 0.
6. While queue not empty
 - a. Remove process P_i from the queue.
 - b. $AT_i = BT_i - TQ$
 - c. If $AT_i < BT_i$, add P_i again in the queue, update WT and goto step 6.
 - d. Calculate waiting time (WT) and turnaround time (TT) for process.
7. Stop

4) PRIORITY SCHEDULING:

Always the process having highest priority is allocated CPU.

Algorithm:

1. Read no. of processes N.
2. Read burst time (BT) and priority for each process and enter it in the list.
3. Start time (ST) = 0.
4. While queue not empty
 - a. Remove process with the highest priority from the list.
 - b. Use ST and BT to calculate Finish Time (FT) of process.
 - c. Use ST and FT to calculate waiting time (WT) and turnaround time (TT) for process.
5. Stop

1.7 Conclusion:

Thus we can implement scheduling algorithms using different methods.

1.8 Questions:

Q1. Find turnaround time and waiting time when the following jobs are scheduled using FCFS, SJF, RR ($TQ=2$) and Priority. Also draw Gantt chart.

Process	Arrival Time	Burst time
P1	0	3
P2	1	6
P3	2	4
P4	3	2

Q.2 What are advantages and disadvantages of FCFS, SJF, Priority scheduling?

Q.3 What should be value of time quantum in RR?

Signature of Staff with date

EXPERIMENT NO: -06 (Group B)

DATE:-

1.1 Title: Write a program to simulate memory placement strategies –best fit, first fit, next fit and worst fit.

1.2 Aim: To simulate memory placement strategies i.e. best fit, first fit, next fit and worst fit.

1.3 Objectives: 1. To acquire knowledge memory placement strategies
2. To be able to implement memory placement strategies

1.4 Hardware used: Experimental Setup/Instruments/ Devices:

1.5 Software used (if applicable) / Programming Languages Used:

1.6 Theory:-

Why Memory Management is required:

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Fragmentation:

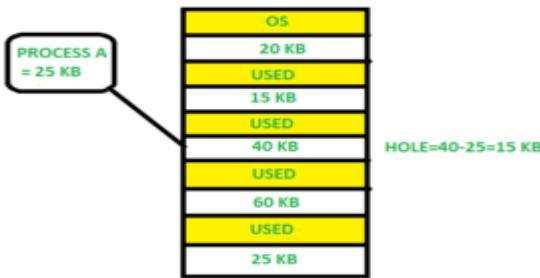
A Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process. To achieve a degree of multiprogramming, we must reduce the waste of memory or fragmentation problem. In operating system two types of fragmentation:

1. Internal Fragmentation
2. External Fragmentation

Memory placement strategies:

1. First fit

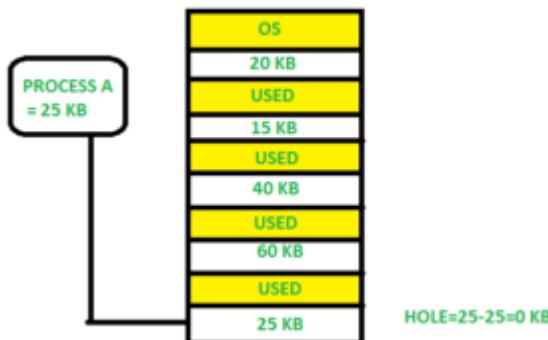
In the first fit, the first available free hole fulfills the requirement of the process allocated.



Here, in this diagram 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

Best fit:-

In the best fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.



Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB).

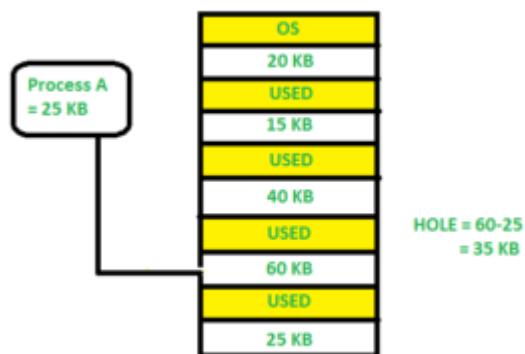
In this method memory utilization is maximum as compared to other memory allocation techniques.

Next fit:

Next fit is another version of First Fit in which memory is searched for empty spaces similar to the first fit memory allocation scheme. Unlike first-fit memory allocation, the only difference between the two is, in the case of next fit, if the search is interrupted in between, the new search is carried out from the last location.

Next fit can also be said as the modified version of the first fit as it starts searching for a free memory as following the first-fit memory allocation scheme. This memory allocation scheme uses a moving pointer which moves along the empty memory slots to search memory for the next fit. The next fit memory allocation method avoids memory allocation always from the beginning of the memory space. The operating system uses a memory allocation algorithm, also known as the scheduling algorithm for this purpose.

Worst fit:- In the worst fit, allocate the largest available hole to process. This method produces the largest leftover hole.



Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

Algorithm/Flowchart:

1. First Fit algorithm/pseudo code

- o Read all required input
- o FOR i<-0 to all jobs 'js'

 - FOR j<-0 to all blocks 'bs'

 - o IF block[j]>=jobs[i]
 - Check jth block is already in use or free
 - Continue and search next free block
 - Otherwise allocate jth block to ith job

 - o Display all job with allocated blocks and fragmentation

2. First Fit algorithm/pseudo code

- o Read all required input
- o FOR i<-0 to all jobs 'js'

 - SET BestInd□ -1
 - FOR j<-0 to all blocks 'bs'

 - o IF block[j]>=jobs[i]
 - IF Block is free and BestInd== -1 THEN SET BestInd□j
 - ELSEIF Block is free and block[BestInd]>block[j] THEN SET BestInd□j
 - ELSE continue with next block

 - Continue and search next free block
 - IF BestInd!= -1 THEN allocate jth block to ith job
 - o Display all job with allocated blocks and fragmentation

3. Worst Fit Algorithm/Pseudo code

- o Read all required input
- o FOR i<-0 to all jobs 'js'

 - SET WstInd□ -1
 - FOR j<-0 to all blocks 'bs'

- o IF block[j]>=jobs[i]
 - IF Block is free and WstInd== -1 THEN SET WstInd□j
 - ELSEIF Block is free and block[WstInd]<block[j] THEN SET WstInd□j
 - ELSE continue with next block
 - Continue and search next free block
 - IF WstInd!= -1 THEN allocate jth block to ith job
 - o Display all job with allocated blocks and fragmentation
4. As above write algorithm of Next Fit strategies

1.7 Conclusion : successfully implemented simulation of memory placement strategies.

1.8 Questions

- Q. 1. Which algorithm is best and why?
- Q. 2. Need of allocating blocks to jobs?
- Q. 3. What is the time taken by each algorithm for execution?

Signature of Staff with date

EXPERIMENT NO: -07 (Group B)**DATE:-****1.1 Title:** Write a program to simulate page replacement algorithm.**1.2 Aim:** To implement page replacement algorithm.**1.3 Objectives:** To understand and implement LRU and Optimal page replacement algorithms.**1.4 Hardware used: Experimental Setup/Instruments/ Devices:****1.5 Software used (if applicable) / Programming Languages Used:****1.6 Theory:-**

Introduction

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to keep the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted. While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms. It is worth noting that the problem of “page replacement” occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency. A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Webpage to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk. In a virtual memory system, pages in main memory may be either clean or dirty.

Concepts

There are several algorithms for page replacement. These algorithms include:

The First-In-First-Out (FIFO) page replacement algorithm.

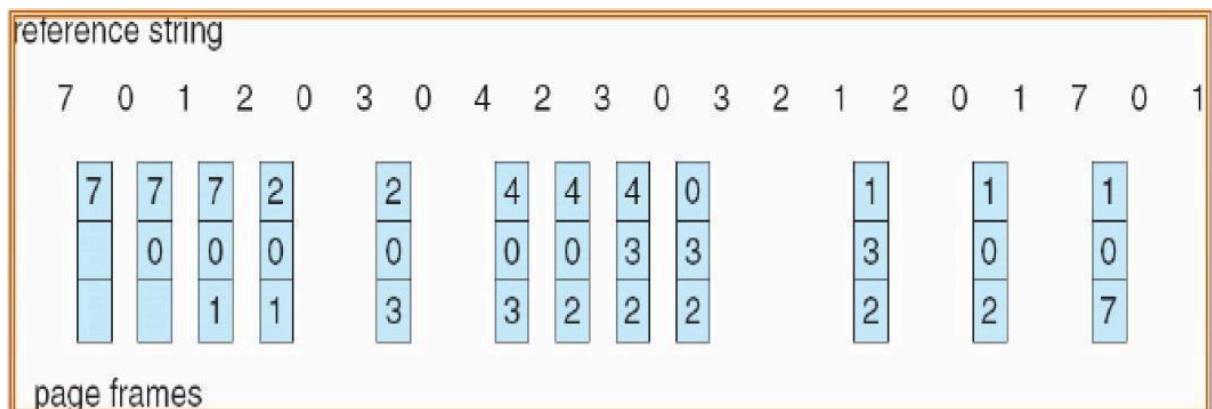
- 1) Least recently used (LRU) page replacement algorithm.
- 2) Optimal (OPT) page replacement algorithm.
- 3) The clock (CLOCK) page replacement algorithm.
- 4) The not recently used (NRU) page replacement algorithm.
- 5) The second chance page replacement algorithm.

A] Least recently used (LRU)

Replaces the page that has been least recently used.

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging.

Example:



No. of page
faults = 12
Algorithm

Pre-conditions: Reference string that will be given as input Post-conditions: Required frames with the final pages.

Steps:

1] Read the initial values:

- Read number of frames
- Read length of the reference string

c. Re

ad

reference

string 2]

Initialize:

a. The array lruf[] is initialized to -1, indicating that frames are empty.

b. the array lrud[] is initialized to 0. It will be used for storing backward distance of pages in the memory.

3] For each page reference i in the reference string

{

a. If the page i is not in the memory and there is an empty frame
then

page i is stored in the empty frame.

b. If the page i is not in the memory and
there is no empty frame then

page i replaces the page with longest backward distance. Backward distance of page i is reset to 0.

c. If the page i is in the memory then Its backward distance is reset to 0.

d. Backward distance of each page (not referenced) is increased by 1.
}

4] Display report.

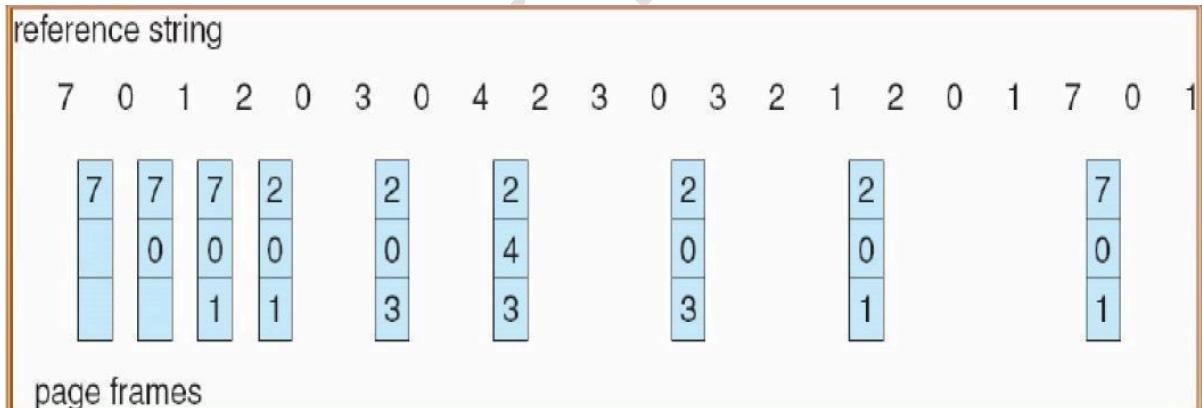
5] End.

B| Optimal Page Replacement (OPT)

Replaces the page that will not be used for long time.

Each page can be labeled with the number of instructions that will be executed before that page is first referenced. The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Algorithms exist that can offer near-optimal performance — the operating system keeps track of all pages referenced by the program, and it uses those data to decide which pages to swap in and out on subsequent runs. This algorithm can offer near-optimal performance, but not on the first run of a program, and only if the program's memory reference pattern is relatively consistent each time it runs.

Example:



No. of page faults: 9

Algorithm

Pre-conditions: Reference string that will be given as input Post-conditions: Required frames with the final pages.

Steps:

- 1] Read the initial values:
 - a. Read number of frames
 - b. Read length of the reference string

```
c.      R
ead
referenc
e string
2]
Initialize
:
a. The array optf[] is initialized to -1, indicating that frames are empty.
b.      the array optd[] is initialized to 0. It will be
used for storing forward distances of pages in the
memory.
3] For each page reference i in the reference string
{
    a.      If the page i is not in the memory and
there is an empty frame then
page i is stored in the empty frame.

    b.      If the page i is not in the memory and
there is no empty frame then
page i replaces the page with longest forward distance.
    c. Forward distance of each page is calculated again.
}
4] Print report.
5] End.
```

1.7 Conclusion:

Thus we have implemented page replacement algorithms

1.8 Questions

- Q. 1 What is page?
- Q.2 What is frame?
- Q.3 What is the need of page replacement?

Signature of Staff with date

EXPERIMENT NO: -08 (Experiment Beyond Syllabus) DATE:-

Title: - Write a program using Lex specifications to implement lexical analysis phase of compiler to count no. of words, lines and characters of given input file.

Aim: To implement lexical analysis phase of compiler to count no. of words, lines and characters of given input file using LEX.

- Objectives:**
- To study LEX
 - To understand Lexical analyzer

Hardware used: Experimental Setup/Instruments/ Devices:**Software used (if applicable) / Programming Languages Used:****Theory:-**

- i. Lex is a program designed to generate scanners also known as tokenizers, which recognize lexical pattern in text.
- ii. Lex is an acronym that stands for “lexical analyzer generator”.
- iii. It is intended primarily for Unix-based system.
- iv. Its main job is to break up an input stream into usable elements.
- v. Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequence called tokens for use as input to other programs such as parsers.

Example:-

```
/*Definition Section*/
%{
#include<stdio.h>
```

```

%}

/** Rule Section */

/*[0-9]+ matches a string of one or more digits*/

[0-9]+ { /* yytext is string containing the
           matched text*/ Printf("Saw an
           integer : %s \n", yylex);

}

%%

/* Code
Section of
C */ int
main(void)

{ /* Call the
   lexer, then
   quit */
yylex();

return 0;
}

```

Built-in variables and functions in Lex :-

1. yytext -> yytext is a pointer to matched string.
2. yyleng -> yyleng is the length of matched string.
3. yyin -> The input file for lex is yyin and defaults to stdin.
4. Yylex() -> yylex is the main entry point for lex.

Implementation:

- i. The declaration section contains one file pointer that points to a text file and integer variable word count, character count and line count which are initialized to zero.
- ii. The rule section defines rule for word, digit and new line.
- iii. The user section defines all the rules that are specified in the rule section.
- iv. For counting, use 'yylen' which is used to calculate length of word, char etc.
- v. In main function display all the counts.

Test Case :- Input :

Let if input text files contains: Lex is a program designed to generate scanners also known as tokenizers 123

Output :

The resulting output:

Number of words are = 12
Number of characters are = 61
Number of digits are = 3
Number of decimal digits are = 1
Number of lines are = 3

Conclusion: Thus, we have used lex and its built-in functions to read text file and count number of words, characters and lines.

Questions:

Q.1 Which are different meta characters used to write regular expression using Lex? Explain with example.

Signature of Staff with date

EXPERIMENT NO: -09

DATE:-

1.1 Title: Implement the mini project (Miniproject should consist of GUI and advanced concepts of programming languages)

M.V.P.S.K.B.T.C.O.