Jayawant Shikshan Prasarak Mandal

T.E. Computer Engineering

Laboratory Practice –I

Laboratory Manual

Laboratory Objectives:

1.To learn system programming tools

2.To learn modern operating system

3. To learn various tachniques,tools,applications in IOT and embedded system.

Experiment Learning Outcome: -

On completion of the course, learners will be able to

• Systems Programming and Operating System

CO1: Implement language translators
CO2: Use tools like LEX and YACC
CO3: Implement internals and functionalities of Operating System

• Internet of Things and Embedded Systems

CO4: Design IoT and Embedded Systems based application
CO5: Develop smart applications using IoT
CO6: Develop IoT applications based on cloud environment

GROUP A

Assignment no.:-1a

TITLE: IMPLEMENTATION OF PASS – I OF TWO PASS ASSEMBLER

OBJECTIVES:

- To study basic translation process of assembly language to machine language.
- To study different data structures required to implement pass I of assembler.

PROBLEM STATEMENT:

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.

SOFTWARE & HARDWARE REQUIREMENTS:

1. 64-bit Open source Linux or its derivative

2. Eclipse

3. JDK

THEORY:

A language translator bridges an execution gap to machine language of computer system. An assembler is a language translator language. Language processing activity consists of two phases, Analysis phase and synthesis phase. Analysis of source program consists of three components, Lexical rules, syntax rules and semantic rules. Lexical rules govern the formation of valid statements in source language.

Semantic rules associate the formation meaning with valid statements of language. Synthesis phase is concerned with construction of target language statements, which have the same meaning as source language statements. This consists of memory allocation and code generation.

WHAT IS AN ASSEMBLER?

An assembler is an translator which takes its input in the form of an assembly language program and produces machine language code as its output

Assembly Language statements: -

There are three types of statements,

1] Imperative - An imperative statement indicates an action to be performed during the execution of assembled program. Each imperative statement usually translates into one machine instruction.

2] Declarative - Declarative statement e.g. DS reserves areas of memory and associates names

with them. DC constructs memory word containing constantscontaining constants.

3] Assembly directives- Assembler directives instruct the assembler to perform certain actions

during assembly of a program, e.g. START<constant> directive indicates that the first word of

the target program generated by assembler should be placed at memory word with address

<constant>

DESIGN OF TWO PASS ASSEMBLER

Pass I: -

1. Separate the symbol, mnemonic opcode and operand fields.

2. Build the symbol table and the literal table.

3. Perform LC processing.

4. Construct the intermediate representation code for every assembly language statement.

Pass II: -

Synthesize the target code by processing the intermediate code generated during pass1

DATA STRUCTURES REQUIRED BY PASS I:

- OPTAB – a table of mnemonic op codes
- Contains mnemonic op code, class and mnemonic info
- Class field indicates whether the op code corresponds to Imperative Statement (IS), Declaration Statement (DL) or Assembler Directive (AD)

For IS, mnemonic info field contains the pair (machine opcode, instruction length)

- SYMTAB - Symbol Table Contains address and length
- LOCCTR - Location Counter
- LITTAB – a table of literals used in the program Contains literal and address
- Literals are allocated addressesstarting with the current value in LC and LC is incremented, appropriately List of hypothetical instructions:

Advanced Assembler Directives :

• LTORG

• ORIGIN

• EQU

LTORG and Literal Pool

The LTORG directive, which stands for ‘origin for literals’, allows a programmer to specify

where literals should be placed. The assembler uses the following scheme for placement of literals: When the use of a literal is seen in a statement, the assembler enters it into a literal

pool unless a matching literal already exists in the pool At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of the literal pool and clears the literal pool. This way, a literal pool would contain all literals used in the program since the start of the program or since the previous LTORG an LTORG statement, the assembler would enter all literals used in the program into a single pool and allocate memory to them when it

encounters the END statement.

Advantages of Literal Pool

• Automatic organization of the literal data into sections that are correctly aligned and

arranged so that minimal space is wasted in the literal pool.

• Assembling of duplicate data into the same area.

ORIGIN Directive

The syntax of this directive is

ORIGIN <address specification>

Where <address specification> is an EQU Directive

The EQU directive has the syntax

<symbol> EQU <address specification>

Where <address specification> is an <operand specification> or <constant>

The EQU statement simply associates the name <symbol> with the address specified by

<address specification>.

However the address in the location counter is not affected.

Algorithm (Pass I of Two – Pass Assembler)

1. LC := 0; (This is a default value)

littab_ptr := 1;

pooltab_ptr := 1;

POOLTABLE[1].first := 1;

2. While the next statement is not an END statement

(a) If a symbol is present in label field then

this_lable := symbol in label field;

Make an entry (this_label, <LC>, __) in SYMTAB.

(b) If an LTORG statement then

(i) If POOLTAB[pooltab_ptr].#literal > 0 then littab_ptr,

POOLTAB[pooltab_ptr].#literals := 0;

(c) If a START or ORIGIN statement then

LC : = value specified in operand field;

(d) If an EQU statement then

(i) this_addr : = value of <address specification>

(ii) Correct the SYMTAB entry for this_lable to (this_label, this_addr , 1)

(e) If a declaration statement then

(i) Invoke the routine whose id is mentioned in the mnemonic info field.

(ii) If the symbol is present in the label field, correct SYMTAB entry for

this_label to (this_label, LC>, Size)

(iii) LC := LC + 1;

(iv) Generate intermediate code for declaration statement.

(f) If an imperative statement then

(i) code : = machine code from mnemonic info field of OPTAB;

(ii) LC := LC + instruction length from length field of OPTAB;

(iii) If Operand is literal then

this_literal : = literal in operand field;

If this_literal does not match any literal in LITTAB then

LITTAB[littab_ptr].value := this_literal;

POOLTAB[pooltab_ptr].#literal = POOLTAB[pooltab_ptr].#literal +1;

littab_ptr := littab_ptr + 1;

else (i.e operand is a symbol)

this_entry := SYMTAB entry number of operand;

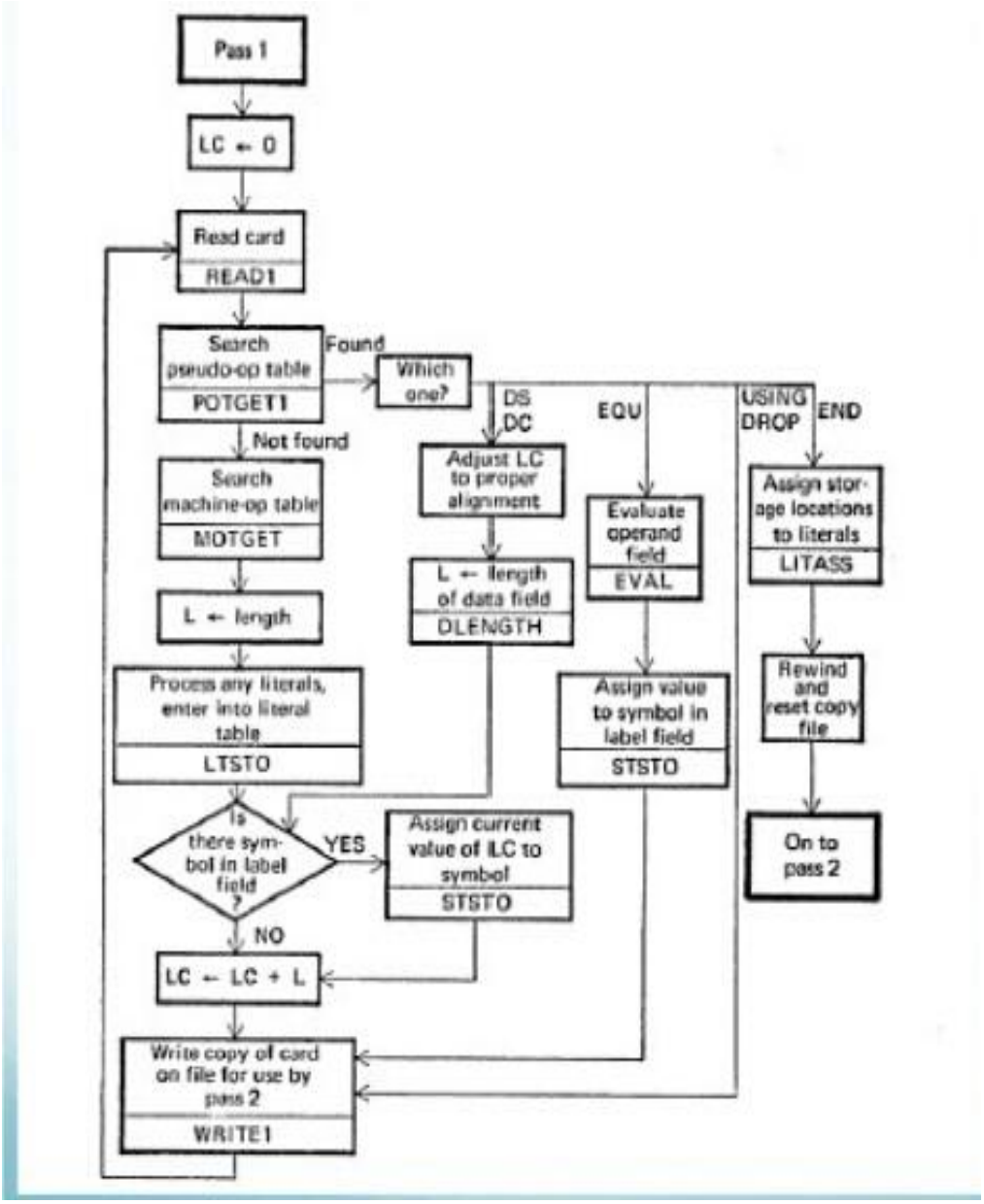
Generate intermediate code for the imperative statement.

3. Processing of END statement

(a) Perform actions (i) – (iii) of Step 2(b)

(b) Generate intermediate code for the END statement.

FLOWCHART FOR PASS 1



```
        START   200             LC
        MOVER   AREG, ='5'      200
        MOVEM   AREG, X         201
L1      MOVER   BREG, ='2'      202
        ORIGIN  L1+3
        LTORG                   205
                                206
NEXT    ADD     AREG,='1'       207
        SUB     BREG,='2'       208
        BC      LT, BACK        209
        LTORG                   210
                                211
BACK    EQU     L1              212
        ORIGIN  NEXT+5
        MULT    CREG,='4'       212
        STOP                    213
X       DS      1               214
        END
```

EXAMPLE:

X DS 1 214

Symbol	Address
X	214
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211
='4'	----

Pool Table
0
2
4

END

Symbol	Address
X	214
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211
='4'	215

Pool Table
0
2
4
5

CONCLUSION: We can design our own Pass I assembler

Assignment no.:-1b

TITLE: Implement Pass II OF TWO PASS ASSEMBLER

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.

OBJECTIVES:

- To study basic translation process of intermediate code to machine language.
- To study and implement pass II of two pass assembler

SOFTWARE & HARDWARE REQUIREMENTS:

1. 64-bit Open source Linux or its derivative
2. Eclipse
3. JDK

THEORY:

Data Structure used by Pass II:

1. OPTAB: A table of mnemonic opcodes and related information.
2. SYMTAB: The symbol table
3. LITTAB: A table of literals used in the program
4. Intermediate code generated by Pass I

5. Output files containing Target code / error listing.

Pass – II of Two Pass Assembler Algorithm

- LC : Location Counter
- littab_ptr : Points to an entry in LITTAB
- Pooltab_ptr : Points to an entry in POOLTAB
- machine_code_buffer : Area for construction code for one statement
- code_area : Area for assembling the target program
- code_area_address : Contains address of code_area

1. code_area_address := address of code_area;

   pooltab_ptr := 1;

   LC := 0;

2. While the next statement is not an END statement

(a) Clear machine_code_buffer;

(b) If an LTORG statement

(i) If POOLTAB[pooltab_ptr].#literal > 0 then

Process literals in the entries LITTAB[POOLTAB[pool_ptr].

first … LITTAB[POOLTAB[pool_ptr+1]-1] similar to

processing of constants in a DC statement. It results in

assembling the literals in machine_code_buffer.

(ii) size := size of memory area required for literals;

(iii) pooltab_ptr := pooltab_ptr + 1;

(c) If a START or ORIGIN statement

(i) LC := value specified in operand field;

ii) size := 0;

(d) If a declaration statement

(i) If a DC statement then

Assemble the constant in machine_code_buffer.

(ii) size := size of the memory area required by the declaration

statement;

(e) If an Imperative statement

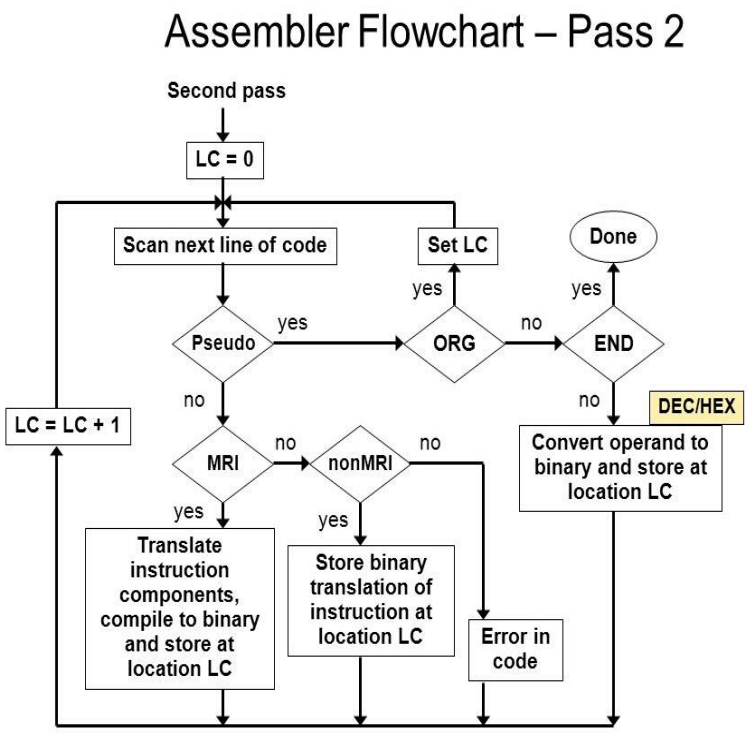
(i) Get address of the operand from its entry in SYMTAB or LITTAB,

(ii) Assemble the instruction in machine_code_buffer.

(iii) size := size of the instruction;

(f) If size ≠ 0 then

(i) Move contents of machine_code_buffer to memory word with the address



code_area_address + <LC>;

(ii) LC := LC + size;

3. Processing of END statement

(a) Perform actions (i) – (iii) of Step 2(b)

(b) Perform actions (i) – (ii) of Step 2(f)

(c) Write code_area into the output file.

Flowchart

CONCLUSION: We can design our own Pass II assembler

Assignment no.:-2a

TITLE: Design suitable data structures and implement pass-I of a two-pass macroprocessor using oop features in java.

OBJECTIVES:
To understand macro facility, features and its use in assembly language programming.
To study how the macrodefinition is processed and how macro call results in the expansion of code.

PROBLEM STATEMENT:
Implement Pass-I of two-pass macroprocessor.

SOFTWARE REQUIRED: 64-bit Open Source Linux or its derivative,jdk

INPUT: Input data as sample program.

OUTPUT: It will generate data structure by scanning all the macro definition.

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

Macro Definition

Macro definition typically appears at the start of a program. Each macro definition is enclosed between a macro header statement and a macro end statement, having mnemonic opcodes MACRO and MEND. Statements included in the macro definition can use formal parameters ; the '&' is prefixed to the name of a formal parameter to differentiate a formal parameter's name from symbolic name.

A macro prototype statement declares the name of the macro and names and kinds of formal parameters.

<macro name> [<formal parameter specification> [,…]]

```
MACRO      -------- Start of definition
INCR       -------- Macro name
A 1,DATA   }
A 2,DATA   } -----Sequence of instructions to be abbreviated
A 3,DATA   }
MEND       -------- End of definition
```

Macro Expansion

The macro processor substitutes the definition for all occurrences of abbreviations (macro call) in the program. Replacement of macro call by corresponding sequence of instructions is called as macro expansion. Macro expansion can be performed by using two kinds of language processors.

Macro Assembler

Macro Preprocessor

A macro assembler performs expansion of each macro call in a program into sequence of assembly statements and also assembles the resulting program.

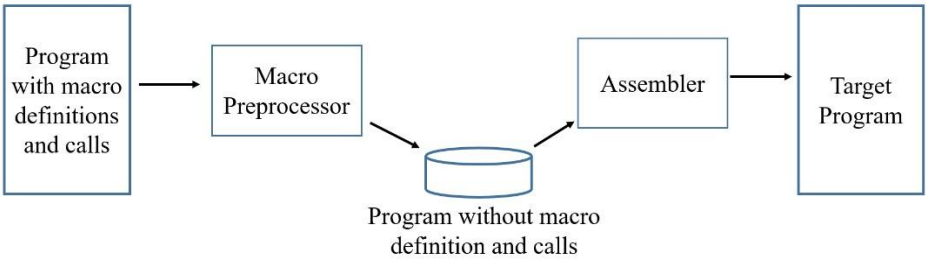
A macro preprocessor merely performs macro expansion of macro calls in a program.

It produces assembly program in which a macro call has been replaced by statements that resulted from its expansion but statements that were not macro calls have been retained in their original form. This program can be assembled using assembler.

Design of a Macro preprocessor

The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definition or call. The program form output by the macro preprocessor would be handed over to an assembler to obtain the target program.

This way, a macro facility is made available independent of an assembler, which makes it both simpler and less expensive to develop.



Design Procedure:-

Tasks involved in macro expansion

Identify macro calls in the program.

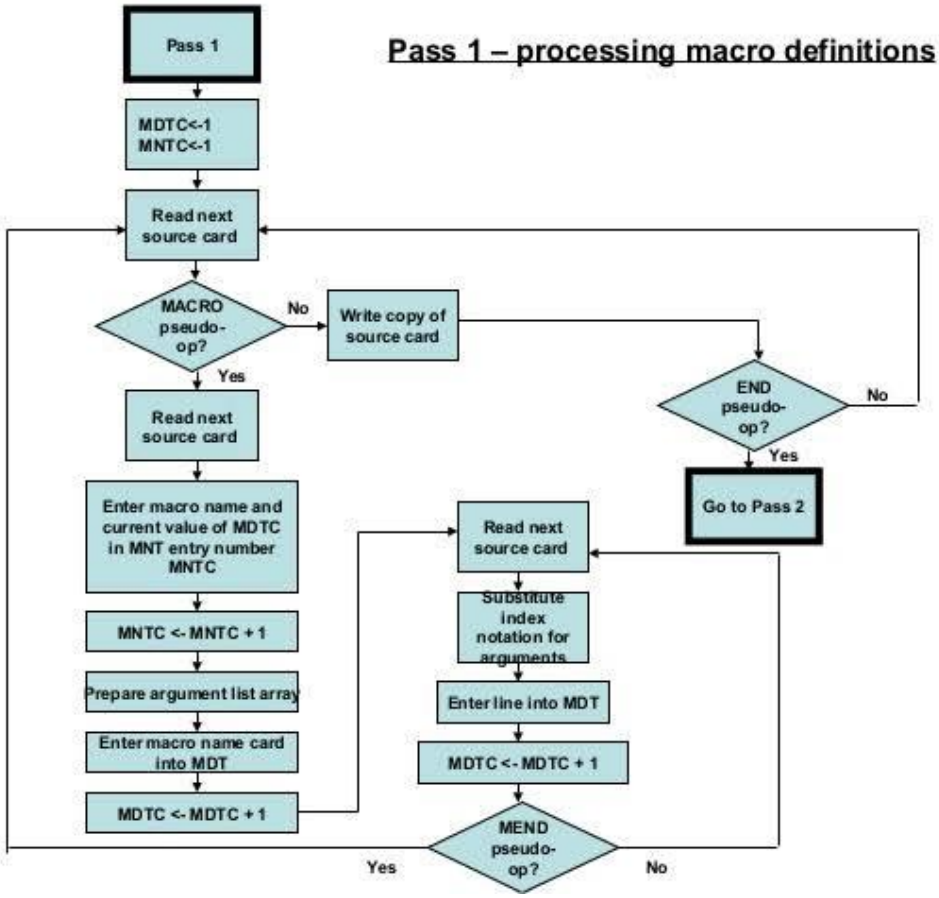
Determine the values of formal parameters.
Maintain the values of expansion time variables declared in a macro.
Organize expansion time control flow.
Finding the statement that defines a specific sequencing symbol.
Perform expansion of a model statement.
Design of Two – Pass Macro Processor
The Two – Pass Macro Processor will have two systematic scans or passes, over the input text, searching first for macro definitions and then for macro calls.
Just as assembler cannot process a reference to a symbol before its definition, so the macro processor cannot process a reference to a symbol before its definition, so the macro cannot expand a macro call before having found and saved macro definition. Thus we need two passes over the input text, one to handle definitions and one to handle calls.

Design of Two – Pass Macro Processor Pass I:
Generate Macro Name Table (MNT)
Generate Macro Definition Table (MDT)
Generate IC i.e. a copy of source code without macro definitions.

Pass II:
Replace every occurrence of macro call with macro definition.

Macro Processor Pass – I Data Structures
The input file (Source Program)
The output file (To be used by Pass – II)
The Macro Definition Table (MDT), used to store the body of the macro definitions.
The Macro Name Table (MNT), used to store the names of defined macros
The Macro Definition Table Counter (MDTC), used to indicate the next available entry in the MDT
The Macro Name Table Counter (MNTC), used to indicate the next available entry in the MNT
The Argument List Array (ALA), used to substitute index markers for dummy arguments before storing a macro definition.

Flowchart:



Data structure Used in pass1 :
1.Macro Definition Table(MDT):Used to store name of macros.

2.Macro Name Table(MNT):The entire macro definition is stored in the MDT.
The index into MDT is stored into MDT.

3.Argument List Array(ALA):The argument list array is used to substitute index markers for formal parameters before storing macro definition in MDT.

4.Macro Name Table Pointer(MNTP):MNTP gives the next available entry in MNT.

5. Macro Definition Table Pointer(MDTP):MDTP gives the next available entry in MDT.

Example:

```
MACRO FIRST
ADD AREG, BREG
MEND
MACRO
SECOND &ARG1,&ARG2
ADD AREG,&ARG1
SUB BREG,&ARG2
MEND
MACRO          THIRD &ARG1=DATA3,&ARG2=DATA1
ADD AREG,&ARG1
ADD BREG,&AGR2
SUB BREG,&ARG1
 MEND
```

```
START
FIRST
:                                        SECOND DATA1,DATA2
:
:
THIRD DATA1,DATA3
:
DATA1  DS 3
DATA2  DS 2
DATA3  DC  '3'
END
```

PASS I Output

MNT:

Sr.No	Macro Name	Index
1	FIRST	1
2	SECOND	4
3	THIRD	8

ALA for SECOND MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	
2	&ARG2	

ALA for THIRD MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	
2	&ARG2	

MDT :

Sr. No	Label	Mnemonic	Operands	
1		FIRST		
2		ADD	AREG	BREG
3		MEND		
4		SECOND	&ARG1	&ARG2
5		ADD	AREG	#1
6		SUB	BREG	#2
7		MEND		
8		THIRD	&ARG1=DATA3	&ARG2=DATA1
9		ADD	AREG	#1
10		ADD	BREG	#2
11		SUB	BREG	#1
12		MEND		

CONCLUSION:

Assignment no.:-2b

TITLE: Implement Pass-II of two-pass macroprocessor

OBJECTIVES: To understand macro facility, features and its use in assembly language programming.
To study how the macro definition is processed and how macro call results in the expansion of code Implement Pass-I of two-pass macroprocessor.

SOFTWARE REQUIRED: 64-bit Open Source Linux or its derivative,jdk

INPUT: Input data as sample program.

OUTPUT: It will generate Expanded code by scanning the code.

THEORY:
Design Procedure:-
Tasks involved in macro expansion
Identify macro calls in the program.
Determine the values of formal parameters.
Maintain the values of expansion time variables declared in a macro.
Organize expansion time control flow.
Finding the statement that defines a specific sequencing symbol.
Perform expansion of a model statement.
Design of Two – Pass Macro Processor
The Two – Pass Macro Processor will have two systematic scans or passes, over the input text, searching first for macro definitions and then for macro calls.
Just as assembler cannot process a reference to a symbol before its definition, so the macro processor cannot process a reference to a symbol before its definition, so the macro cannot expand a macro call before having found and saved macro definition. Thus we need two passes over the input text, one to handle definitions and one to handle calls.

Design of Two – Pass Macro Processor Pass I:
Generate Macro Name Table (MNT)
Generate Macro Definition Table (MDT)
Generate IC i.e. a copy of source code without macro definitions.

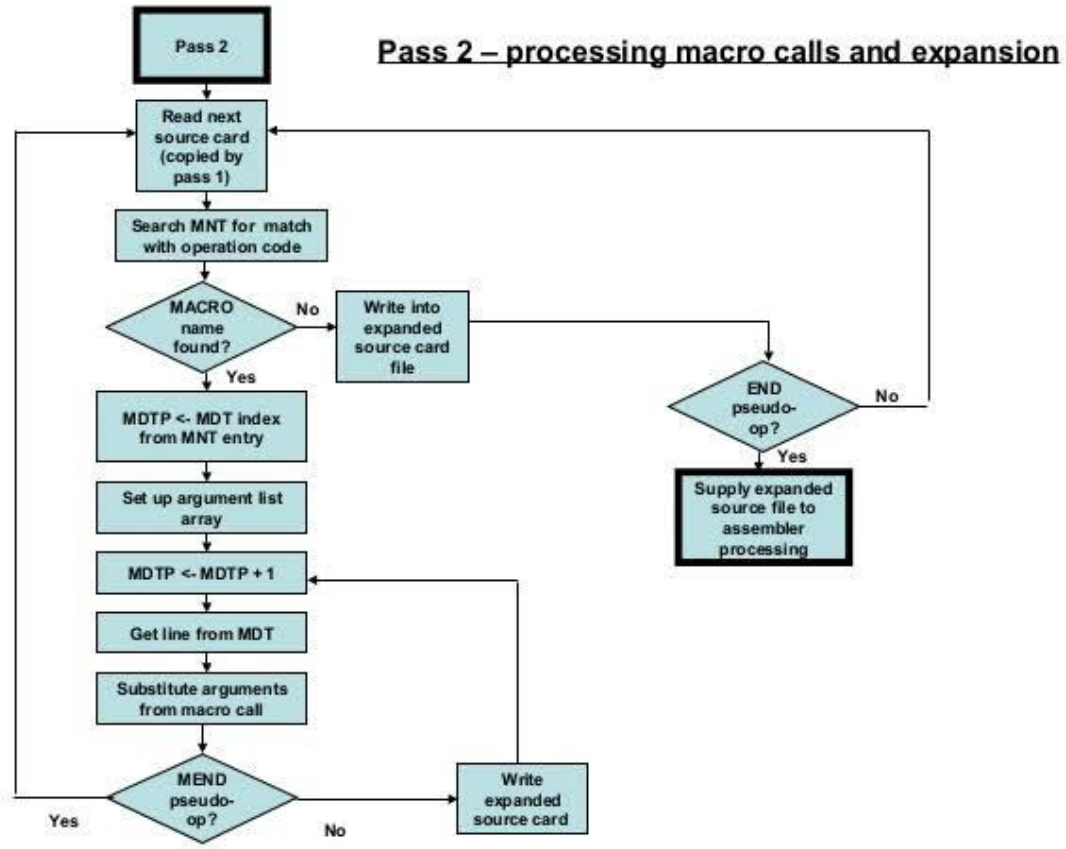
Pass II:
Replace every occurrence of macro call with macro definition.

Macro Pass – II (Macro Calls & Expansion)
The algorithm for Pass- II tests mnemonic of each line to see if it is a name in the MNT. When a call is found, the call processor sets a pointer, the Macro Definition Table Pointer (MDTP), to the corresponding macro definition stored in MDT.
The initial value of MDT is obtained from the "MDT index" field of the MNT entry.
The macro expander prepares the Argument List Array (ALA) consisting of a table of dummy argument indices and corresponding arguments to the call.
As each successive line is read, the values from the argument list are substituted for dummy argument indices in the macro definition. Reading of the MEND line in the MDT terminates expansion of the macro, and scanning continues from the input file. When the END is encountered, the expanded source program is transferred to the assembler for further processing.

Macro Processor Pass – II Data Structures

The input file (Output of Pass - I)
The output file (To store Target Program)
The Macro Definition Table (MDT), created by Pass – I
The Macro Name Table (MNT) gives number of entries in MNT.
The Macro Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro expansion
The Argument List Array (ALA), used to substitute macro call arguments for the index markers in the stored macro definition

FLOWCHART:



Example:
START
ADD AREG,BREG
ADD AREG,DATA2
SUB BREG,DATA1
ADD AREG,DATA3
ADD BREG,DATA1
SUB AREG,DATA3
DATA1 DS 3
DATA2 DS 2
DATA3 DC '3'
END

PASS I OUTPUT:

ALA FOR SECOND MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	
2	&ARG2	

ALA FOR THIRD MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	
2	&ARG2	

PASS II OUTPUT:

ALA FOR SECOND MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	DATA1
2	&ARG2	DATA2

ALA FOR THIRD MACRO:

Sr.No	Formal Args	Actual Args
1	&ARG1	DATA1
2	&ARG2	DATA3

Conclusion:

Group:- B

Assignment no.:-3

TITLE: Process Scheduling Algorithms.

OBJECTIVES:

1. To understand cpu scheduling algorithms.

PROBLEM STATEMENT:

Write a java program to implement scheduling algorithm like FCFS, SJF, Priority, Round Robin.

SOFTWARE REQUIRED: Linux Operating Systems, GCC, java editor, jdk s/w

INPUT: Arrival time, Burst time of process.

OUTPUT: It will generate the total turn around time and waiting time.

THEORY:

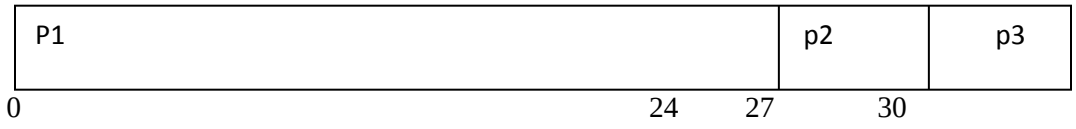
CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

First come, first served:

With this scheme, the process that requests the CPU first is allocated the CPU first. Consider the following set of processes that arrive at time 0, burst time is given.

Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order p1,p2,p3 and served in FCFS order, then Gantt Chart is :



Thus, the average waiting time is (0+24+27)/3 = 17 ms.

Shortest Job First:

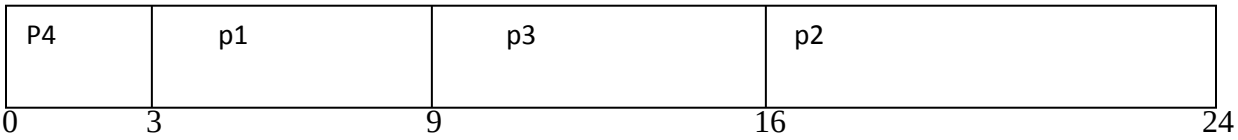
This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the two process have the same length next CPU burst, FCFS scheduling is used to break the tie.

Consider the following set of processes.

Process	Burst Time
P1	6
P2	8

P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt Chart :



Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 ms.

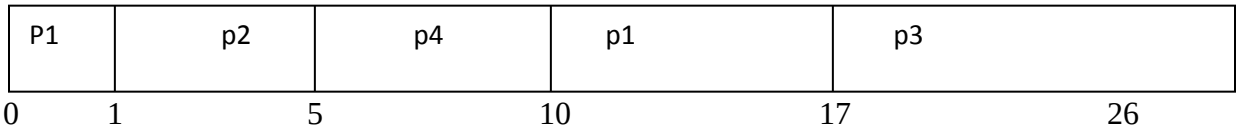
Preemptive SJF

When new process arrives at the ready queue while previous process is executing, the new process may have a shorter next CPU burst than what is left of the currently executing process, preemptive algo. will preempt the currently executing process.

E.g.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Corresponding Gantt Chart :



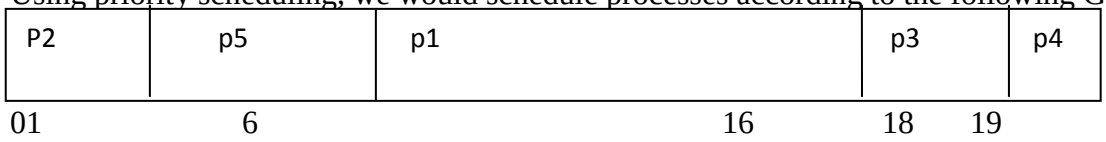
The avg. waiting time for this example is ((10 – 1)+(1 – 1) + (17 – 2) + (5 – 3))/4 = 6.5 ms.

Priority Scheduling :

A prority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

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

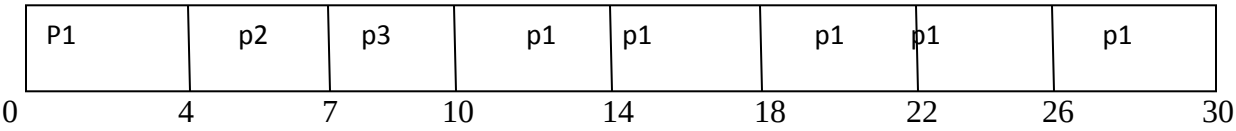
Using priority scheduling, we would schedule processes according to the following Gantt chart



The avg. waiting time is 8.2 msec

Round Robin Scheduling :

Process	Burst Time
P1	24
P2	3
P3	3



The avg. waiting time is 17/3 = 5.66 ms.

CONCLUSION : Thus we have studied CPU scheduling algorithms.

Assignment no.:-4

TITLE: Page replacement policies

OBJECTIVES:

1. To understand and compare page replacement methods.

PROBLEM STATEMENT:

Write a java program to implement page replacement policies like LRU (Least Recently Used), Optimal and FIFO.

SOFTWARE REQUIRED:

Linux Operating Systems, GCC, java editor, jdk s/w

INPUT:Page string, size of memory frame.

OUTPUT:

It will generate the page faults , page hits , fault ratio and hit ratio.

THEORY:

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous.

In paged memory management each job's address space is divided into equal no of pieces called as pages and physical memory is divided into pieces of same size called as blocks or frames.

Whenever there is a page reference for which the page needed is not in memory that event is called as page fault.

Suppose all page frames are occupied and new page has to be brought in due to page fault, in such case we have to make space in memory for this new page by replacing any existing page. There are several algo. or policies for page replacement.

FIFO page replacement:

When a page must be replaced, the oldest page is chosen.

Consider the reference string 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5 with memory of three references

Ref. String	4	3	2	1	4	3	5	4	3	2	1	5
	4	4	4	1	1	1	5	5	5	5	5	5
		3	3	3	4	4	4	4	4	2	2	2
			2	2	2	3	3	3	3	3	1	1
Fault	+	+	+	+	+	+	+			+	+	

Page Fault = 9 **hit ratio** = 3/12 , fault ratio=9/12

Algo.is affected by the **Belady's anomaly** (the page fault rate may increase as the number of allocated frames increases.)

Optimal page replacement:

The basic idea of this algo is to replace the page that will not be used for the longest period of time. It has the lowest page fault of all algo. and will never suffer from the Belady'sanamoly.

Consider the string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with three memory frames.

Ref. string	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Fault	+	+	+	+		+		+			+			+				+		

Page fault = 9 **Hit ratio** = 11/20 , fault ratio=9/20

This algo is difficult to implement because it requires further knowledge of reference string

LRU page replacement:

We will replace the page that has not been used for the longest period of time. This algo.also never suffers from Belady'sanamoly.

Consider the string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with three memory frames

Ref. string	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
			1	1	3	3	3	2	2	2	2	2	2	2	2	2	2	7	7	7
Fault	+	+	+	+		+		+			+			+				+		

Page fault = 9 **Hit ratio** = 11/20, fault ratio=9/20

CONCLUSION : Thus studied page replacement algorithm.