Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology, Pune-37

Department Of Computer Engineering

Lab Manual

Class: - TE **Branch: - Computer & Info. Technology**

Prepared By: - Prof. D.J.Joshi & A.A.Bhilare Year: -2012-13

Required H/W and S/W: - P-III, DOS/Windows, Linux, C

Contents

ASSIGNMENT NO-1

Title- Implementation of Macro Processor

Aim- Implementation of Macro Processor. Following cases to be considered while implementation

a) Macro without any parameters

b) Macro with Positional Parameters

c) Macro with Key word parameters

d) Macro with positional and keyword parameters.

(Conditional expansion, nested macro implementation not expected)

Pre Lab-

- Concept of assembly language.
- Concept of preprocessing.
- Process of two pass macro processor.

Theory-

An assembly language macro facility is to extend the set of operations provided in an assembly language.

In order that programmers can repeat identical parts of their program, macro facility can be used. This permits the programmer to define an abbreviation for a part of program & use this abbreviation in the program. This abbreviation is treated as macro definition & saved by the macro processor. For all occurrences the abbreviation i.e. macro call, macro processor substitutes the definition.

Macro definition part:

It consists of

1. Macro Prototype Statement - This declares the name of macro & types of parameters like positional, keyword or mixing of both.
2. Model statement - It is statement from which assembly language statement is generated during macro expansion.
3. Preprocessor Statement - It is used to perform auxiliary function during macro expansion.

Macro Call & Expansion:
The operation defined by a macro can be used by writing a macro name in the mnemonic field and its operand in the operand field. Appearance of the macro name in the mnemonic field leads to a macro call. Macro call replaces such statements by sequence of statement comprising the macro. This is known as macro expansion.

Design of Two pass macro processor-

Data structures required for macro definition processing:

1. Macro Name Table [MNT]
   Fields - Index, Name of Macro, MDTP (Macro Definition Table Pointer).

2. Argument List Array Table [ALA]
   Fields –Index, Parameter Name

3. Macro Definition Table [MDT]
   Model Statements present between MACRO and MEND keywords are stored.

   Fields-Statement no, definition

4. Counters MNTC, ALAC, and MDTC for MNT, ALA and MDT respectively.

Sample Input -

The assembly language program with macro definitions & macro calls

```
MACRO

INCR &MEM_VAL, &INCR_VAL, &REG

MOVER   &REG, &MEM_VAL

ADD      &REG, &INCR_VAL

MOVEM  &REG, &MEM_VAL

MEND

START

MOVER AREG, B

ADD AREG, N

INCR A, B, AREG

MOVEM AREG, N

MOVEM BREG, N
```

```
MOVEM A, N
  A    DS   5
  B    DS   1
  N    DS   1
END
```

Sample output -

Macro Name Table

Index	Name	MDTP
1	INCR	0

Argument List Array table

Index	Formal Argument	Actual Argument
1	&MEM_VAL	A
2	&INCR_VAL	B
3	&REG	AREG

Macro Definition Table

Index	Statements of the macros
0	INCR &MEM_VAL, &INCR_VAL, &REG
1	MOVER #3,#1
2	ADD #3,#2
3	MOVEM #3,#1
4	MEND

Expanded code with no macro definition & macro calls

```
        START

MOVER AREG, B

ADD AREG, N

MOVER AREG, A

ADD AREG, B

MOVEM AREG, A

MOVEM AREG, N

MOVEM BREG, N

MOVEM A, N

  A    DS   5

  B    DS   1

  N    DS   1

END
```

Conclusion-

With the help of implementation of this two pass macro processor we demonstrated that the macros are used to provide a program generation facility through macro expansion which is lexical expansion or semantic expansion.

Lexical expansion implies replacement of formal parameters by corresponding actual parameters.

Semantic expansion implies generation of instructions tailored to the requirements of a specific usage.

Advancement-

The program can be extended for

- Conditional macro expansion
- Macro definition within macro
- Macro call within a macro

Post Lab-

Following objectives are met

- 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.

Viva Questions-

1. Define the term macro.
2. Distinguish between macro and a subroutine
3. Define and Distinguish between parameters that can be used in macros.
4. State various tables used in processing the macro.
5. Explain the role of stack in nested macros.

ASSIGNMENT NO-2

Title- Implementation of Nested Macros

Nested macro calls refer to the macro calls within the macros.
•A macro is available within the other macro definitions
.•In the scenario where a macro call occurs, which contains another macro call, the macro processor generates the nested macro definitions text and places it on the input stack.
•The definition of the macro is then scanned and the macro process or compiles it.
•The following example can make you understand the nested macrocall.

```
MACRO
SUB 1 &PAR
 L  1,&PAR
A   1,=F'2'
ST1,&PAR
MEND
MACRO
SUBST &PAR1, &PAR2,&PAR3
SUB1&PAR1
SUB2&PAR2
SUB3&PAR3
MEND
```

It can be easily noticed from the example that the definition of the macro 'SUBST' contains three separate calls to a previously defined macro 'SUB1'.
•The definition of the macro 'SUB1' has shortened the length of the definition of the macro 'SUBST'.

The following code describes how to implement a nested macro call:
Source:

```
:
MACRO
SUB1        &PAR
L           1,&PAR
A           1,='F2'
ST          1, &PAR
MEND
MACRO
SUBST          &PAR1,&PAR2,&PAR3
SUB1          &PAR1        DATA1 DC F'5'
SUB1          &PAR2        DATA2 DC F'10'
SUB1          &PAR3        DATA3 DC F'15'
MEND
 :                                  :
                                    :
                                    :
SUBST DATA1, DATA2, DATA3
```

Expanded Source(Level 1)

Expansion of SUBST	Expansion source(Level 2)
	Expansion of SUB1
:	:
:	:
:	:
SUB1 DATA1	L 1,DATA1
	A 1,=F'2'
	ST 1,DATA1
SUB1 DATA2	L 1,DATA2
	A 1,=F'2'
	ST 1,DATA2
SUB1 DATA2	L 1,DATA3
	A 1,=F'2'
	ST 1,DATA3
	:
	:
	:
	DATA1 DC F'5'
	DATA2 DC F'10'
	DATA3 DC F'15'

ASSIGNMENT NO-3

Title- Two Pass Assembler

Aim-Implementation of Two Pass assembler with hypothetical Instruction set.

Instruction set should include all types of assembly language statements such as Imperative, Declarative and Assembler Directive. While designing stress should be given on

A) How efficiently Mnemonic opcode table could be implemented so as to enable faster retrieval on op-code.

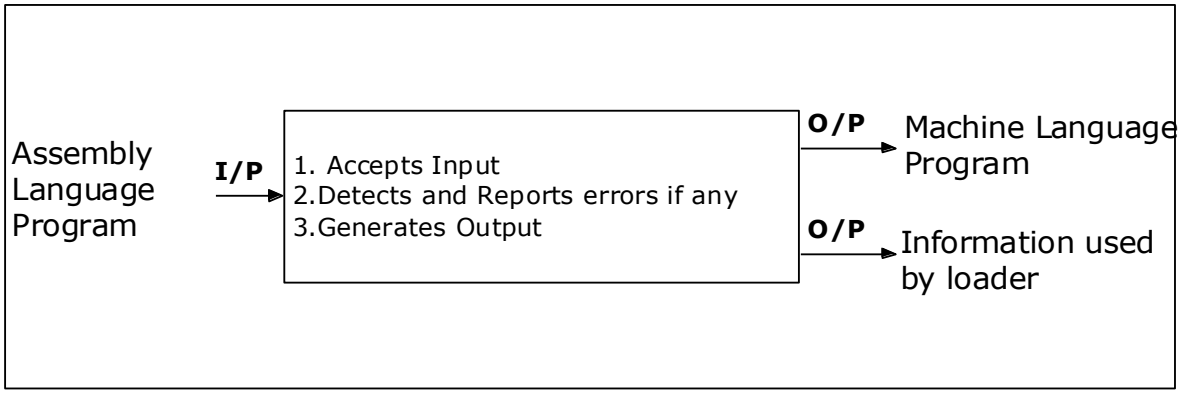
B) Implementation of symbol table for faster retrieval.

OBJECTIVES:

1. To understand how assembler work?.
2. To understand what actions are taken in pass 1 and pass 2?
3. To learn how symbol table is created by assembler?
4. How and when it used MOT to generate object code?
5. To learn what is the format of object file?

Theory-

Assembler: It creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities.



There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code. The assembler must at least be able to

determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.

Design of Two pass assembler:

Task performed by the passes of a two pass assembler are as follows:

Pass I =>

1) Separate the symbol, mnemonic opcode and operand fields.
2) Build the symbol table.
3) Perform LC processing.
4) Construct intermediate representation.

Pass II =>

Synthesize the target program.

As shown in fig.1.1 the Pass I perform analysis of the source program and synthesis of the intermediate representation while Pass II processes the intermediate representation to synthesize the target program.

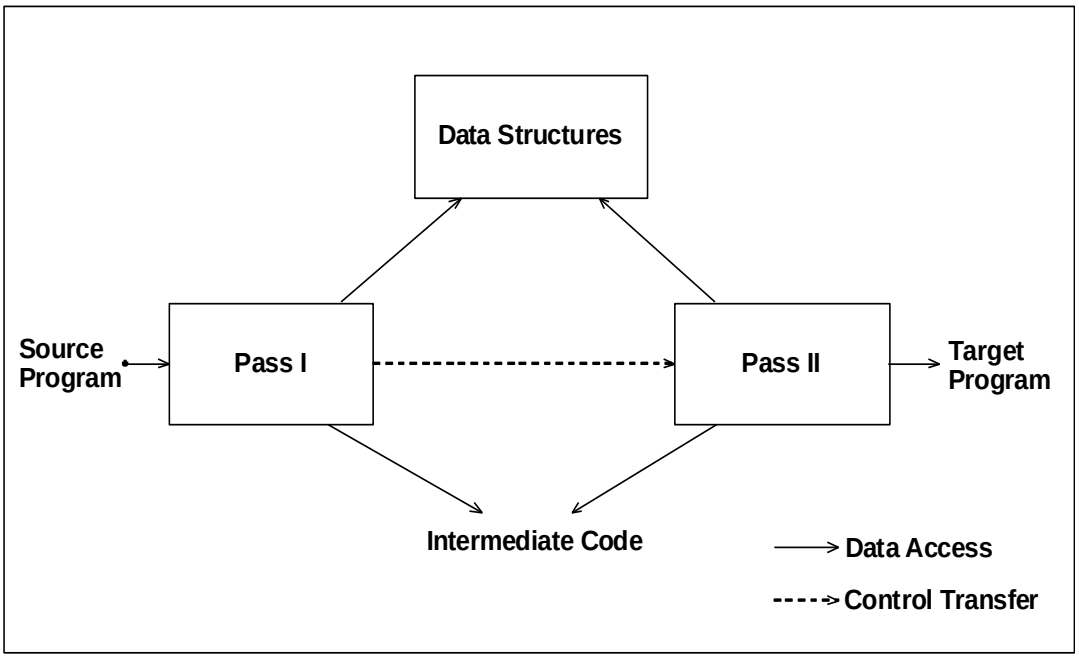


Figure 1.1 Overview of two pass assembly

Required Data structures

1) Opcode Table: Assembler need not to enter information in this table. This table with information is available with the assembler.
   The assembler does not enter information in this table but uses this data structure

   - To determine valid imperative statement.
   - Find length of instruction
   - Fine corresponding machine Opcode.

   Fields of Opcode Table:

   a) Opcode
   b) Machine code
   c) Type of Opcode
   d) Length of Opcode

2) Register Table: This table stores the information about registers supported by system.
   Fields of Register Table:

   a) Register name
   b) Machine constant

3) Symbol Table: This table is used to record information about symbols in program.
   Symbols can be

   - Label appearing on any statement
   - Constant declared by 'DC' statement
   - Storage variable declared by 'DS' statement

   This table is created by Pass-I and used by Pass-II of 2-pass assembler.

   Fields of Symbol Table:

   a) Symbol no
   b) Symbol name
   c) Symbol address
   d) Symbol length

4) Literal Table: This table is used to store the information about literals. This table is created by Pass-I and used by Pass-II of 2-pass assembler.
   Fields of Literal Table:

a) Literal no
b) Literal
c) Literal address

5) Literal Pool Table: This table stores information about literal pools in the assembly program. No of literal pools is always 1 greater than number of LTORG statements in assembly program.
This table is created by Pass-I and used by Pass-II of 2-pass assembler.

Pool table can be implemented as one dimensional array where index of array indicates literal pool no and POOLTABLE[x] indicates literal number of starting literal in literal pool 'x'

Representation of Intermediate Code

Format of Intermediate Code is as follows:

Address	Opcode	Operand

1) Address:
Address can be specified or determined by using location counter

2) Opcode:
It has two parts:

a) Type of Opcode i.e. IS, DL, AD
b) Machine code

3) Operand:
It has two parts:

a) First Operand i.e. Register operand with IC (Machine constant)

b) Second Operand i.e
Symbol operand with IC (S, Symbol no) or

Literal Operand with IC (L, Literal no) or

Constant Operand with IC (C, Constant value)

Representation of Machine Code

Format of Machine Code is as follows:

Opcode	Operand

1) Opcode:
It consists of Machine code of the opcode

2) Operand:
It has two parts:

a) First Operand i.e. Register operand with machine code (Machine constant)
b) Second Operand i.e
Symbol operand with machine code as its address or

Literal Operand with machine code as its address or

Constant Operand with machine code as its value

Following things are used

1. **Instruction set: -** This is an instruction set for a hypothetical machine.

Opcode	**Mnemonic**	**Remark**
01	STOP	Stop execution of program
02	ADD	They take two operands and result is stored in First operand. They are setting a condition code
03	SUB	
04	MULT	
05	MOVER	Register<= memory
06	MOVEM	Memory <= Register
07	DIV	Division operation
08	READ	Read from a memory address to a REGA
09	PRINT	Display the contents of address

Instruction set

2. Registers used: - This machine uses only four registers AREG, BREG, CREG and DREG. Every register has an address as shown in Register Address Table (RAT).

Register	Address
AREG	000

BREG	001
CREG	010
DREG	011

Register Address Table

3. **Data structures: -** Following data structures are used by assembler
    a. **OPTAB: -** Operation table. Class **IS** (imperative statement=00), Declaration statement DL=01, and Assembler directive AD=10

mnemonic	class	info	length
STOP	00	01	1
ADD	00	02	3
SUB	00	03	3
MULT	00	04	3
MOVER	00	05	3
MOVEM	00	06	3
DIV	00	07	3
READ	00	08	2
PRINT	00	09	2
DC	01	00	----
DS	01	01	----
START	10	01	---
END	10	02	---
EQU	10	03	----

OPTAB (Operation Table)

Symbol Table (ST): -

Symbol	Address
LOOP	45
…..	…..

Literal table (LT):-

Literal name	Address
=’5’	67
…..	…..

Implementation Issues of Data structures

1. Symbol table and literal tables are implemented as linked lists.
2. OPTAB, RAT is stored on hard disk in the text files.
3. After PASS –II symbol table and Literal table must be deleted.
4. Order of searching these data structures is OPTAB, RAT, ST and LT.

5. sample source code: -

```
START 101
MOVER AREG, ONE
MOVER BREG, TWO
ADD AREG, BREG
MOVEM AREG, SUM
PRINT SUM
```

6. Format of OUTPUT file (Object Module): - Assembler has to generate a file, which need to be linked and loaded for execution. Such an output file is called as Object file or object module. In this file format relocation table and Linking information is stored; but for simplicity we will consider format of out put file as shown below.

Size of program
Address of an instruction from where actual execution begins
Object code

1. **Size of program:-** can be found by (Value of LC – Address in START), thus it is better to call pass2 with this parameter.
2. **Address of an instruction from where actual execution begins :-** can be found after START
3. **Object code :-** can be copied from code area.

ALGORITHM 1 (PASS 1)

1. Set LC=0.
2. While next statement is not an END then

a). If symbol definition is found then make an entry in ST.

b). If START then LC= Value given after start. Generate intermediate code

(10,01) (C, Address specified)

c). If EQU statement then add symbol to ST which is on RHS of EQU and

assign address of it as address of symbol on LHS of EQU.

d) If declaration statement then find size of memory area required and code

associated with the statement and generate intermediate code (01,CODE)

and increment LC =LC + size.

e). IF an imperative statement

1. LC=LC + length
2. Generate intermediate code (00, CODE)

3. If operand is Literal then enter it into LT. IF IT is symbol then enter it into ST.
4. If END statement generate intermediate code (10, 02) and goto pass-II.

OUT PUT of PASS-I

Source program	**LC**	**Intermediate file**
START 101	LC=0 (default)	(10, 01) (C, 101)
MOVER AREG, ONE	LC=101	(00,05) AREG, ONE
MOVER BREG, TWO	LC=104	(00, 05) BREG, TWO
ADD AREG, BREG	LC=107	(00, 02) AREG, BREG
MOVEM AREG, SUM	LC=110	(00, 06) AREG, SUM
PRINT SUM	LC=113	(00,09) SUM
ONE DC '2'	LC=115	(01,00) (C,2)
TWO DC '3'	LC=116	(01,00) (C,3)
SUM DS 1	LC=117	(01, 01) (C, 1)
END		(10, 02)

ALGORITHM 2 (PASS 2)

1. Define a code area (to some max value of program size), code buffer (to max size of any instruction) and LC=0 and variable SIZE=0;

2. While next statement is not END i.e. (10, 02) then do

a). clear the buffer

b) If START statement i.e. (10, 01) then

i) LC=value after C in (C,101)
ii) SIZE=0

c) If a declaration statement i.e. (01, XX) (XX- means any) then

ii) If DC statement i.e (01,00) then assemble the constant in buffer
iii) SIZE= size of memory area required to accommodate constant/space specified in DC/ DS statement

d). If an imperative statement i.e (00,XX)

i) Get operand address from ST/LT/RAT
ii) Assemble instruction in buffer like 00,XX, 000, 101
iii) SIZE= size of this instruction

e) If SIZE is not equal to zero then

i). Then copy the contents of buffer to the

address (code-area-address)+ LC

ii) LC=LC+SIZE

3. If statement is END i.e (10, 02) then
   i) Open a file "object" and
   ii) Copy the contents of code area into the output file
4. Close the output file and Free the space occupied by
   i) ST, LT
   ii) Buffer
   iii) Code area

Output of Pass –II

Source program	**Intermediate file**	**LC**	**Out Put File**
START 101	(10, 01) (C, 101)	LC=0 (default)	-------
MOVER AREG, ONE	(00,05) AREG, ONE	LC=101	00 05 000 115
MOVER BREG, TWO	(00, 05) BREG, TWO	LC=104	00 05 001 116
ADD AREG, BREG	(00, 02) AREG, BREG	LC=107	00 02 000 001
MOVEM AREG, SUM	(00, 06) AREG, SUM	LC=110	00 06 000 117

PRINT SUM	(00,09) SUM	LC=113	00 09 108
ONE DC '2'	(01,00) (C,2)	LC=115	2
TWO DC '3'	(01,00) (C,3)	LC=116	3
SUM DS 1	(01, 01) (C, 1)	LC=117	<space>
END	(10, 02)		-------

TEST CONDITIONS.

1. Give input as the sample source code and check the outputs of pass 1 and pass 2.
2. Cross check the values of LC.
3. Try to write another source code for this hypothetical instruction set and cross check out put of pass 1 and pass 2.

APPLICATIONS.

1. Assembler is used in translation of assembly language program to object code.

2. Single pass assembler is used when efficient assembling is required and multi pass assembler is used when less memory is utilized.

FAQs.

1. When multi pass assemblers are better than single pass?
2. When single pass assembler is better than multi pass?
3. Why assembler has to maintain LC?
4. Why the output of pass 1 is called "*intermediate code*"?
5. Which of the following table are constructed by assemble and which of them are used by assembler? (ST, LT, OPTAB)
6. Why it is necessary to destroy ST and LT after processing of a source file?
7. When assembler will throw an error "*undefined symbol*" ?
8. In pass 2 how to define maximum program size?
9. Which instructions are having SIZE=0 in pass 2?
10. What will happen when a symbol and mnemonic is same?

ASSIGNMENT NO-4

Title:-Symbol Table generation for *.c file

Compilers that produce an executable (or the representation of an executable in object module format) as opposed to a program in an intermediate language (and, in fact, for optimization purposes, all compilers) need to make use of a symbol table .

The symbol table records information about the identifiers in the source program such as their name, type, no. of dimensions, space assignment, etc.

To illustrate the use of symbol tables, let's consider a simple c program :

```
#include<stdio.h>

void main()

{

int a,b,c;

char d;

float f;

c=a+b;

printf("%d",c)

label:

printf("%c",d)

end:

}
```

This file is given as an input to our program for generating symbol table.

Our Program should filter all the variables and the labels and store their symbol size and address.

Algorithm

1.Read the input file line by line

2.If any literal found then add it in the symbol table

3.literal should be added with the symbol size and address of the literal.

4.while finding literals consider only variables and labels ,ignore all the keywords and reserved words in ‘c’.

The output should look like this

Literal	size	address
a	2	0
b	2	2
c	2	4
d	1	6
f	4	7

label

end

ASSIGNMENT NO-5

Title-Screen Editor

Aim- Implement Screen Editor with following Features:

1) Open an existing file
2) Create and Save the current file.
3) All cursor movements up, down, left, right arrow keys
4) Delete and backspace keys.

Pre Lab-

- Student should be familiar with computer graphics concepts and its implementation in C.

Theory-

Editors are program entry and editing tool through which programs are entered into the computer. It allows creation, update and deletion of items in a file.

A typical editor's GUI has a menu bar, a set of buttons on various toolbars, other controls and finally a large "canvas" area for display and direct manipulation of the document.

Text editors come in following form:

1) Line editors
2) Stream editors
3) Screen editors
4) Word processors
5) Structure editors

Line editor is a text editor computer program that is oriented around lines. The line editor supports editing operations where basic unit is a line.

Example: UNIX ed editor

Stream editor views the entire text as a stream of characters.

Screen editor uses the 'What-You-See-Is-What-You-Get' principle in editor design. It displays a screenful of text at a time where text is conceptually split into screens.

Word Processors are document editors with additional features to produce well formatted hardcopy output.

Structure editors can be used to edit hierarchical or marked up text, computer programs, diagrams, chemical formulas, and any other type of content with clear and well-defined structure.

Structure editing predominates when content is graphical and textual representations are awkward.

Example: CAD systems and PowerPoint.

Design of an Editor-

The fundamental functions in editing are traveling, editing, viewing and display.

Traveling implies movement within the text.

Viewing implies formatting the text in a manner desired by user. This is an abstract view, independent of the physical characteristics of an IO device.

Display component maps this view into physical characteristics of display device being used.

Editing allows changes in text.

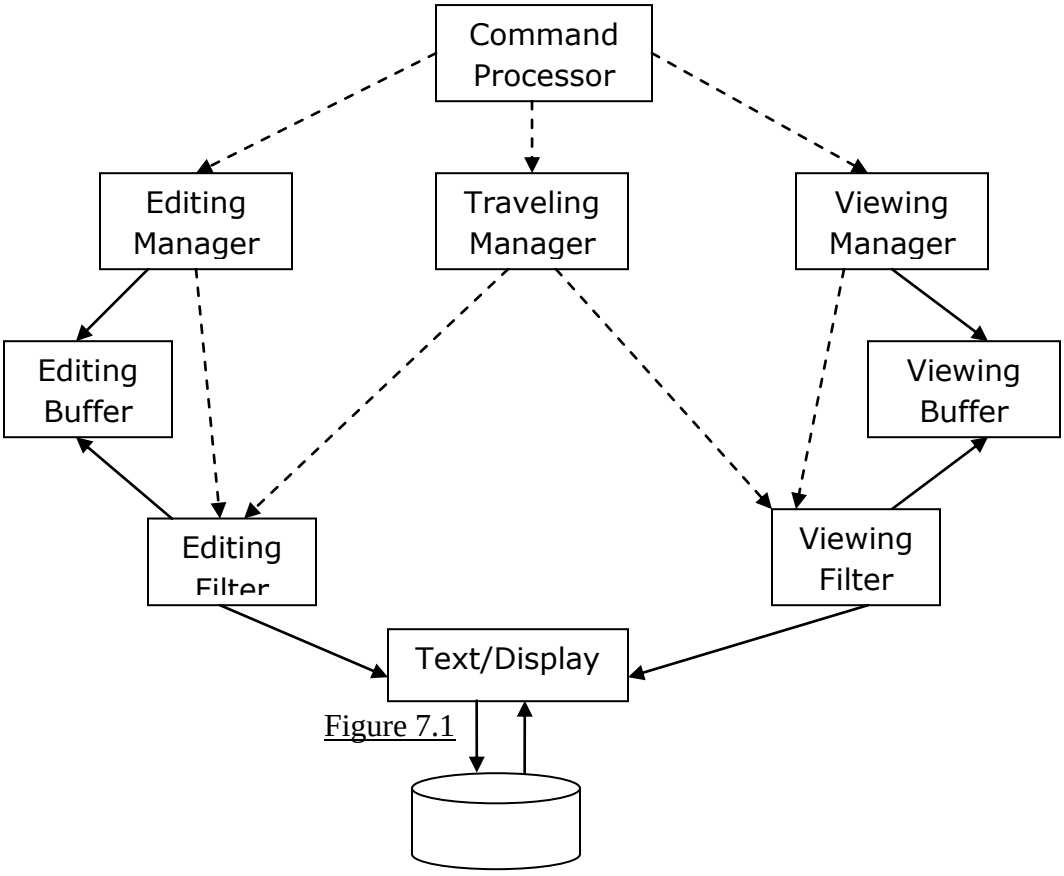


Figure 7.1

Figure 7.1 illustrates the schematic of a simple editor. For a given position of the editing context, the editing and viewing filters operate on the internal form of text to prepare the forms suitable for editing and viewing. These forms are put in the editing and viewing buffers respectively. The viewing-and-display manager makes provision for appropriate display of this text. When the cursor position changes; the filters operate on a new portion of text to update the contents of the buffers. As editing is performed, the editing filter reflects the changes into the internal form and updates the contents of the viewing buffer.

ALGORITHM

1. Use an array of character that can store a line full of text (of size 80 chars)
2. Ask the user which file to newly create or open an existing file.
3. After opening a file editor must allow user for appending a new line at the end of opened file, allow him to delete a specified word, allow adding a given word after a specified word of a line, and allow reading an entire line by line. Program has to extensively use array of characters.
4. When editor program is executed it must provide following alternatives to user
    1. Open an existing file
    2. Open anew file
    3. Opening a file in appending mode
    4. Deleting a file

TEST CONDITIONS.

1. Test for appending a new line to a file.
2. Test for deleting a word from specified file
3. Checking saving of file.
4. Opening and editing of an existing file
5. Reading of a complete file.

APPLICATIONS.

1. It is used as an editor along with Operating system.
2. Used to create, update and delete files.

Conclusion

The following features of screen editor are studied with the help of this implementation

1. Text is conceptually split into screen
2. The user has full control over entire terminal where screen is displayed
3. The user can move cursor over screen for editing operations
4. The user can bring delete a character by pressing delete or backspace key

Advancement-

The program can be implemented by using video memory.

Post Lab-

Following objectives are met

- To understand the role of Screen Editor in language processing

Viva Questions-

1. Define Screen Editor.
2. What are the different categories of editors?
3. What is difference between line editor and screen editor?
4. Explain design of screen editor.
5. Why a line editor is called a “line” editor?
6. What is the difference between a line and screen editor?
7. What OS system calls are used for editors?
8. When line editor is useful than screen editor?
9. Given an example of line editor over UNIX.

Experiment Number: 06

TITLE: Simulation Of Linker.

OBJECTIVES:

1. To understand what is Linker?.
2. To understand How Linker does linking?
3. To learn how to do Linking?

PRE REQUISITE: - Theoretical knowledge of assembly Langauge.

THEORY:

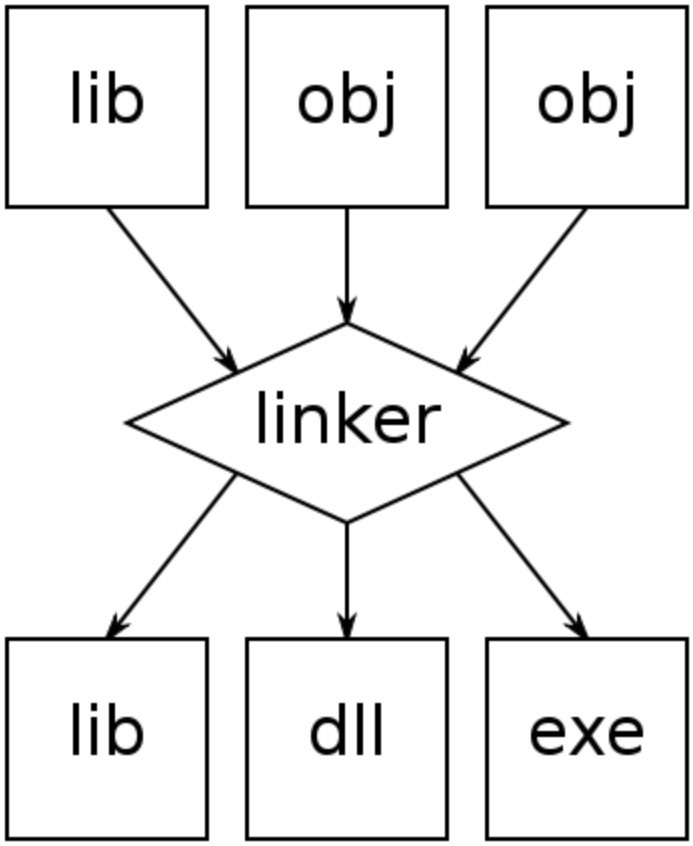
Computer programs typically comprise several parts or modules; all these parts/modules need not be contained within a single object file, and in such case refer to each other by means of symbols. Typically, an object file can contain three kinds of symbols:

- defined symbols, which allow it to be called by other modules,
- undefined symbols, which call the other modules where these symbols are defined, and
- local symbols, used internally within the object file to facilitate relocation.

When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along.

Linkers can take objects from a collection called a *library*. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default. This may involve *relocating* code that assumes a specific base address.

The linker also takes care of arranging the objects in a program's address spaceto another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores.



The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory — every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

ALGORITHM:

1. Create .txt or any other file.
2. Enter data which consist of extern variables also.
3. Write a code to link these files.
4. Perform the linking of files.
5. Check memory requirements while liking for each variable.

FAQs (Frequently Asked Questions)

1. What is Linker?
2. How Linker performs Linking?
3. What kind of files are accepted by Linker for Linking?
4. Is there any difference between Linker and Relocator?if yes. Explain.
5. The executable output by the linker may need another relocation pass. Why?

Experiment Number: 07

TITLE: Simulation Of Loader.

OBJECTIVES:

1. To understand what is Loader?.
2. To understand How Loader works?
3. To learn how to Execute Loader?

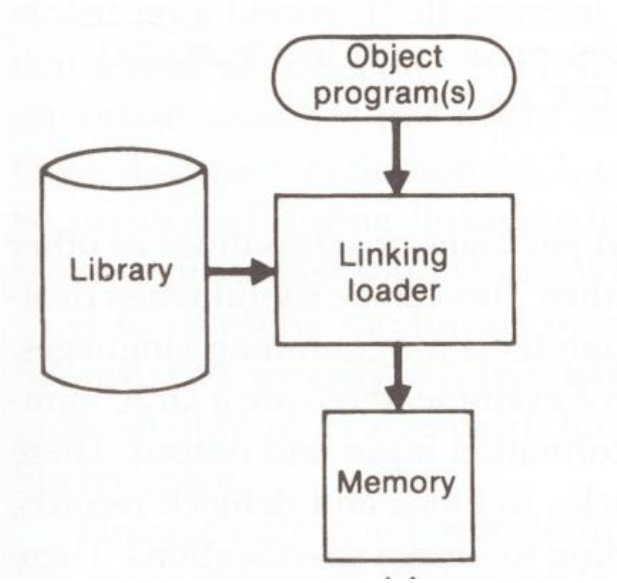
PRE REQUISITE: - Theoretical knowledge of assembly Language.

THEORY:

In computing, a **loader** is the part of an operating system that is responsible for loading programs. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

All operating systems that support program loading have loaders, apart from systems where code executes directly from ROM or in the case of highly specialized computer systems that only have a fixed set of specialised programs.

In many operating systems the loader is permanently resident in memory, although some operating systems that support virtual memory may allow the loader to be located in a region of memory that is pageable.



In the case of operating systems that support virtual memory, the loader may not actually copy the contents of executable files into memory, but rather may simply declare to the virtual memory subsystem that there is a mapping between a region of memory allocated to contain the running program's code and the contents of the associated executable file. (See memory-mapped file.) The virtual memory subsystem is then made aware that pages with that region of memory need to be filled on demand if and when program execution actually hits those areas of unfilled memory. This may mean parts of a program's code are not actually copied into memory until they are actually used, and unused code may never be loaded into memory at all.

In Unix, the loader is the handler for the system call execve().The Unix loader's tasks include:

1. validation (permissions, memory requirements etc.);
2. copying the program image from the disk into main memory;
3. copying the command-line arguments on the stack;
4. initializing registers (e.g., the stack pointer);
5. jumping to the program entry point (_start).

ALGORITHM:

1. Create different .txt or any other files.
2. Write a program to load them.
3. Load the file through code.
4. Generate the output.
5. Check the memory status and address.

FAQs (Frequently Asked Questions)

1. What is Loader?
2. How Loader works?
3. What kind of files are accepted by Loader?
4. Is there any difference between Linker and Loader?if yes. Explain.
5. List out the Functions of Loader?

Experiment Number: 08

TITLE: Design Lex specifications for the tokens – keywords, identifiers, numbers, operators, white spaces.

OBJECTIVES:

1. To understand how a lexical analyser work?.
2. To understand different types of tokens viz. (identifiers, Punctuation symbols) ?
3. Learning to use LEX tool on linux?

PRE REQUISITE: - Theoretical knowledge of lexical analyzer, automatic lexical analyzer generation.

THEORY:

This is a simple tutorial that tells how to use FLEX (Fast LEXical analyzer), which is a LEX tool supplied with Linux as a free software.

1. A first program **sample1.flex** is as follows that recognizes only two keywords viz. begin and end (remember that lex programs are case sensitive) and file extension .flex is optional but it is recommended to use to remember which file is associated with LEX.

```
%{

#include<stdio.h>

%}

%%

begin printf("begin keyword found");

end printf("End keyword found");

%%
```

How to execute this file ? Execute the following commands at command promt

1. $ flex sample1.flex
    - Execution of this command will generate a file lex.yy.c in your present working directory that can be checked
    - By using $ ls command.
2. $ gcc lex.yy.c –l f l
    - gcc is GNU compiler collection used to compile lex.yy.c file and the option –lfl adds linking libraries to the
    - compiled file a.out
    - For executing file a.out use next step
3. $ ./a.out
    - program will wait for you to type and when keywords begin and end are typed then program prints a message that keyword is found
    - A sample output session with this program is given below. The session can be stopped with ctrl-D.

```
begin

keyword begin received

end

end keyword received

END

END ( shows case sensitive)

BEGIN

BEGIN

i am happy (unrecognized things are echoed as it is)

i am happy
```

2. Second program is **sample2.flex**. It takes into account keywords, identifiers and punctuation symbols.

```
%{

#include<stdio.h>

%}

%%

char printf("keyword begin found");

end printf("keyword end found");

if printf("keyword if found");

else printf("keyword else found");

int printf("keyword ");

float printf("keyword");

[a-zA-Z][a-zA-Z0-9]*    printf("IDENTIFIER ");


\"   printf("QUOTE ");

\{   printf("OPENING OF BRACE ");

\}   printf("END OF BRACE ");

;    printf("SEMICOLON ");


%%
```

3. Write a program which include all the keyword of C language which are given below along with some punctual symbols of C.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

ALGORITHM

1. Write a LEX program using any text editing tool over Linux give the extension .flex to it.
2. Compile it using command $ flex abc.. flex
   Execution of this command will generate a file lex.yy.c in your present working directory that can be checked by using $ ls command.
3. Compile it for C using $ gcc lex. yy. c –l f l . We will get a. out as output and executable file.
4. To execute this program use $ ./a.out
5. Then type some tokens and non tokens and check the response of program.

APPLICATIONS.

1. It is used to test validity of a token, so it is important and first phase of compiler.
2. It can also be used to find some “keyword” in given text document.

FAQs.

1. What are the content of file lex.yy.c?
2. What is long form of “Flex”?
3. Why “-l f l ” is used to compile the source file of flex?
4. What is the output of program flex?
5. What is interpretation of regular expression [a-zA-Z0-9]* ?
6. If an unrecognizable string is given as input what will be the output of lexical analyzer?

Experiment Number: 09

TITLE: Writing and using a Dynamic Linking Library (DLL).

OBJECTIVES:

1. To understand creation and use of DLL.
2. How DLLs are used to extend capability of OS?
3. How to compile a DLL source?.

PRE REQUISITE: - Theoretical knowledge of DLL and dynamic linking and loading.

THEORY:

Consider following tutorial for understanding creating and using a DLL

1. What is Static linking and Dynamic linking ?

The libraries can be linked to the executable in two ways . one is static linking and the other is dynamic linking. Static linking is the method used to combine the library routines with the application. When building the program , the linker will search all the libraries for the unresolved functions called in the program and if found, it will copy the corresponding object code from the library and append it to the executable. The linker will keep searching until there are no more unresolved function references. This is the most common method for linking libraries.This method boasts of ease of implementation and speed but will generate bulky executable. 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. This method yields an executable with small footprint , but have to compromise on speed a little.

The idea of a plugin is familiar to users of Internet browsers. Plugins are downloaded from the Web and typically provide enhanced support for audio, video, and special effects within browsers. In general, plugins provide new functions to an existing application without altering the original application.

DLLs are program functions that are known to an application when the application is designed and built. The application's main program is designed with a framework or backplane that optionally loads the required dlls at runtime where the dlls are in files separate from the main application on the disk. This packaging and dynamic loading provides a flexible upgrade, maintenance, and licensing strategy.

Linux ships with thousands of commands and applications that all require at least the libc library functions. If the libc functions were packaged with all programs, thousands of copies of the same functions would be on the disk. Rather than waste the disk space, Linux builds these applications to use a single system-wide copy of the commonly required system libraries. Linux goes even further. Each process that requires a common system library function uses a single system-wide copy that is loaded once into memory and shared.

In Linux, plugins and dlls are implemented as dynamic libraries. The remainder of this article is an example of using dynamic libraries to change an application after the application is running.

Linux dynamic linking

Applications in Linux are linked to an external function in one of two ways: either *statically linked at build time*, with static libraries (`lib*.a`) and having the library code include in the application's executable file, or *dynamically linked at runtime* with shared libraries (`lib*.so`). The dynamic libraries are mapped into the application execution memory by the dynamic linking loader. Before the application is started, the dynamic linking loader maps the required shared object libraries into the application's memory or uses system shared objects and resolves the required external references for the application. Now the application is ready to run.

As an example, here is a small program that demonstrates the default use of dynamic libraries in Linux:

```
 main()
{
   printf("Hello world
");
}
```

When compiled with gcc hello.c, an executable file named `a.out` is created. Using the Linux command `ldd a.out`, which prints shared library dependencies, the required shared libraries are:

```
        libc.so.6 => /lib/libc.so.6 (0x4001d000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

The same dynamic linking loader is used to map a dll into the application's memory after it is running. The application controls which dynamic libraries are loaded and which functions in the libraries are called by using Linux dynamic loader routines to perform the loading and linking and to return the addresses of the required entry points.

Linux dll functions

Linux provides four library functions (`dlopen`, `dlerror`, `dlsym`, `dlclose`), one include file (`dlfcn.h`), and two shared libraries (static `library libdl.a` and dynamic `library libdl.so`) to support the dynamic linking loader. The library function are:

- *dlopen* opens and maps into memory the shared objects file and returns a handle
- *dlsym* return a pointer to the requested entry point
- *dlerror* returns either NULL or a pointer to an ASCII string describing the most recent error
- *dlclose* closes the handle and unmaps the shared objects

The dynamic linking loader routine dlopen needs to find the shared object file in the filesystem to open the file and create the handle. There are four ways to specify the file's location:

- Absolute file path in the `dlopen call`
- In the directories specified in the LD_LIBRARY_PATH environmental variable
- In the list of libraries specified in /etc/ld.so.cache
- In /usr/lib and then /lib

A dll example: small C program and dlTest

The dynamic linking loader example program is a small C program designed to exercise the dl routines. Based on the one C program everyone has written, it prints "Hello World" to the console. The original message printed is "HeLlO WoRlD". The test links to two functions that print the message again: the first time in all uppercase characters and then again in lowercase characters.

This is the program's outline:

1. The dll include file `dlfcn.h` and required variables are defined. The variables needed, at a minimum, are:
    - Handle to the shared library file
    - Pointer to the mapped function entry point
    - Pointer to error strings

2. The original message, "HeLlO WoRlD", is printed.

3. The shared object file for the UPPERCASE dll is opened by `dlopen` and the handle is returned using the absolute path "/home/dlTest/UPPERCASE.so" and the option RTLD_LAZY.
    - Option RTLD_LAZY postpones resolving the dll's external reference until the dll is executed.
    - Option RTLD_NOW resolves all dll external references before `dlopen` returns.

4. The entry point printUPPERCASE address is returned by `dlsym`.
5. printUPPERCASE is called and the modified message, "HELLO WORLD", is printed.
6. The handle to UPPERCASE.so is closed by `dlclose`, and the dll is unmapped from memory.
7. The shared object file, lowercase.so, for the lowercase dll is opened by `dlopen` and the handle returned using a relative path based on the environmental variable LD_LIBRARY_PATH to search for the shared object.
8. The entry point printLowercase address is returned by `dlsym`.
9. printLowercase is called and the modified message, "hello world", is printed.
10. The handle to lowercase.so is closed by `dlclose` and the dll is unmapped from memory.

Note that after each call to `dlopen`, `dlsym`, or `dlclose`, a call is made to `dlerror` to get the last error and the error string is printed. Here is a test run of dlTest:

```
        dlTest  2-Original message
HeLlO WoRlD
        dlTest  3-Open Library with absolute path return-(null)-
        dlTest  4-Find symbol printUPPERCASE return-(null)-
HELLO WORLD
        dlTest  5-printUPPERCASE return-(null)-
        dlTest  6-Close handle return-(null)-
        dlTest  7-Open Library with relative path return-(null)-
        dlTest  8-Find symbol printLowercase return-(null)-
hello world
        dlTest  9-printLowercase return-(null)-
        dlTest 10-Close handle return-(null)-
```

The complete source lists for dlTest.c, UPPERCASE.c and lowercase.c are under Listings later in this article.

Building dlTest

Enabling runtime dynamic linking is a three-step process:

1. Compiling the dll as position-independent code
2. Creating the dll shared object file
3. Compile the main program and link with the dl library

The gcc commands to compile the UPPERCASE.c and lowercase.c include the option -fpic. Options -fpic and -fPIC cause code generation to be position-independent, which is required to recreate a shared object library. The -fPIC options produces position-independent code, which is enabled for large offsets. The second gcc command for UPPERCASE.o and lowercase.o is passed the -shared option, which produces a shared object file, a *.so, suitable for dynamic linking.

The ksh script used to compile and execute dltest is:

```
#!/bin/ksh
#  Build shared library
#
#set -x
clear

#
#  Shared library for dlopen absolute path test
#
if [ -f UPPERCASE.o ]; then rm UPPERCASE.o
fi
gcc  -c -fpic UPPERCASE.c
if [ -f UPPERCASE.so ]; then rm UPPERCASE.so
fi
gcc -shared -lc  -o UPPERCASE.so  UPPERCASE.o

#
#  Shared library for dlopen relative path test
#
export LD_LIBRARY_PATH=`pwd`
if [ -f lowercase.o ]; then rm lowercase.o
fi
gcc  -c -fpic lowercase.c
if [ -f lowercase.so ]; then rm lowercase.so
fi
gcc -shared -lc  -o lowercase.so  lowercase.o

#
#  Rebuild test program
#
if [ -f dlTest ]; then rm dlTest
fi
gcc -o dlTest dlTest.c -ldl
echo Current LD_LIBRARY_PATH=$LD_LIBRARY_PATH
dlTest
```

Summary

Creating shared objects that can be dynamically linked at runtime to an application on a Linux system is a simple exercise. The application gains access to the shared objects by using function calls dlopen, dlsym, and dlclose to the dynamic linking loader. Any errors are returned in strings, by dlerror, which describe the last error encountered by a dl function. At runtime, the main

application finds the shared object libraries using either an absolute path or a path relative to LD_LIBRARY_PATH and requests the address of the needed dll entry points. Indirect function calls are made to the dlls as needed, and finally the handle to the shared objects is closed and the objects are unmapped from memory and made unavailable.

The shared objects are compiled with an additional option, -fpic or -fPIC, to generate position-independent code and placed into a shared object library with the -shared option.

Shared object libraries and the dynamic linking loader available in Linux provides additional capabilities to applications. It reduces the size of executable files on the disk and in memory. Optional application functions can be loaded as needed, defect can be fixed without rebuilding the complete application, and third-party plugins can included in the application.

Listings (the application and dlls)

```
/*************************************************************/
/*     Test Linux Dynamic Function Loading                   */
/*                                                           */
/*     void       *dlopen(const char *filename, int flag)
       */
/*             Opens dynamic library and return handle       */
/*                                                           */
/*     const char *dlerror(void)                             */
/*             Returns string describing the last error.
       */
/*                                                           */
/*     void       *dlsym(void *handle, char *symbol)         */
/*             Return pointer to symbol's load point.
       */
/*             If symbol is undefined, NULL is returned.
       */
/*                                                           */
/*     int        dlclose (void *handle)
       */
/*             Close the dynamic library handle.
       */
/*                                                           */
/*                                                           */
/*                                                           */
/*************************************************************/
#include<stdio.h>
#include       <stdlib.h>

/*                                                     */
/* 1-dll include file and variables   */
/*                                                     */
#include       <dlfcn.h>
void  *FunctionLib;           /*  Handle to shared lib file  */
int   (*Function)();          /*  Pointer to loaded routine  */
const char *dlError;          /*  Pointer to error string          */
```

```
main( argc, argv )
{
  int   rc;                            /*  return codes
       */
  char HelloMessage[] = "HeLlO WoRlD\n";

/*                                                                 */
/* 2-print the original message                                          */
/*                                                                 */
  printf("     dlTest  2-Original message \n");
  printf("%s", HelloMessage);

/*                                                 */
/*  3-Open Dynamic Loadable Libary with absolute path      */
/*                                                */
  FunctionLib = dlopen("/home/dlTest/UPPERCASE.so",RTLD_LAZY);
  dlError = dlerror();
  printf("     dlTest  3-Open Library with absolute path return-%s- \n",
dlError);
  if( dlError ) exit(1);

/*                                                                 */
/* 4-Find the first loaded function   */
/*                                                                 */
  Function    = dlsym( FunctionLib, "printUPPERCASE");
  dlError = dlerror();
  printf("     dlTest  4-Find symbol printUPPERCASE return-%s- \n", dlError);
  if( dlError ) exit(1);

/*                                                                 */
/* 5-Execute the first loaded function                             */
/*                                                                 */
  rc = (*Function)( HelloMessage );
  printf("     dlTest  5-printUPPERCASE return-%s- \n", dlError);

/*                                                                 */
/* 6-Close the shared library handle                                     */
/* Note:  after the dlclose, "printUPPERCASE" is not loaded              */
/*                                                                 */
  rc = dlclose(FunctionLib);
  dlError = dlerror();
  printf("     dlTest  6-Close handle return-%s-\n",dlError);
  if( rc ) exit(1);


/*                                                                 */
/*  7-Open Dynamic Loadable Libary using LD_LIBRARY path                 */
/*                                                                 */
  FunctionLib = dlopen("lowercase.so",RTLD_LAZY);
  dlError = dlerror();
  printf("     dlTest  7-Open Library with relative path return-%s- \n",
dlError);
  if( dlError ) exit(1);

/*                                                                 */
/* 8-Find the second loaded function                               */
/*                                                                 */
```

```
  Function    = dlsym( FunctionLib, "printLowercase");
  dlError = dlerror();
  printf("     dlTest  8-Find symbol printLowercase return-%s- \n", dlError);
  if( dlError ) exit(1);

/*                                                                  */
/* 8-execute the second loaded function                                 */
/*                                                                  */
  rc = (*Function)( HelloMessage );
  printf("     dlTest  9-printLowercase return-%s- \n", dlError);

/*                                                                  */
/* 10-Close the shared library handle                               */
/*                                                                  */
  rc = dlclose(FunctionLib);
  dlError = dlerror();
  printf("     dlTest 10-Close handle return-%s-\n",dlError);
  if( rc ) exit(1);

  return(0);

}
/*************************************************/
/*      Function to print input string as UPPER case.          */
/*      Returns 1.
*/
/********************************************** */

int printUPPERCASE ( inLine )
char inLine[];
{
   char UPstring[256];
   char *inptr, *outptr;

   inptr = inLine;
   outptr = UPstring;
   while ( *inptr != '\0' )
      *outptr++ = toupper(*inptr++);
  *outptr++ = '\0';
   printf(UPstring);
   return(1);
}
```

```
/*****************************************/
/*     Function to print input string as lower case.      */
/*     Returns 2.                                         */
/***************************************** */
int printLowercase( inLine )
char inLine[];
{
   char lowstring[256];
   char *inptr, *outptr;
   inptr = inLine;
   outptr = lowstring;
   while ( *inptr != '' )
      *outptr++ = tolower(*inptr++);
  *outptr++ = '';
   printf(lowstring);
   return(2);
}
```

Experiment Number: 10

TITLE: Use of different debugger tools.

OBJECTIVES:

1. To understand ,what is debugger tools?.
2. How Debugger tools are used?

PRE REQUISITE: - Theoretical knowledge of Debugging.

THEORY:

Introduction:

A debugger or debugging tool is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation—full or partial simulation—to limit this impact.

A "crash" happens when the program cannot normally continue because of a programming bug. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the program "crashes" or reaches a preset condition, the debugger typically shows the location in the original code if it is a source-level debugger or symbolic debugger, commonly now seen in integrated development environments. If it is a low-level debugger or a machine-language debugger it shows the line in the disassembly (unless it also has online access to the original source code and can display the appropriate section of code from the assembly or compilation).

Features:

Typically, debuggers also offer more sophisticated functions such as running a program step by step (**single-stepping** or program animation), stopping (**breaking**) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of variables. Some debuggers have the ability to modify program state while it is running. It may also be possible to continue execution at a different location in the program to bypass a crash or logical error.

The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general verification tool, test coverage and performance analyzer, especially if instruction path lengths are shown.

Most mainstream debugging engines, such as gdb and dbx, provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, program animation, and visualization features. Some early mainframe debuggers such as Oliver and SIMON provided this same functionality for the IBM System/360 and later operating systems, as long ago as the 1970s.

Hardware support for debugging:

Most modern microprocessors have at least one of these features in their CPU design to make debugging easier:

- Hardware support for single-stepping a program, such as the trap flag.
- An instruction set that meets the Popek and Goldberg virtualization requirements makes it easier to write debugger software that runs on the same CPU as the software being debugged; such a CPU can execute the inner loops of the program under test at full speed, and still remain under debugger control.
- In-System Programming allows an external hardware debugger to reprogram a system under test (for example, adding or removing instruction breakpoints). Many systems with such ISP support also have other hardware debug support.
- Hardware support for code and data breakpoints, such as address comparators and data value comparators or, with considerably more work involved, page fault hardware.
- JTAG access to hardware debug interfaces such as those on ARM architecture processors or using the Nexus command set. Processors used in embedded systems typically have extensive JTAG debug support.
- Microcontrollers with as few as six pins need to use low pin-count substitutes for JTAG, such as BDM, Spy-Bi-Wire, or debug WIRE on the Atmel AVR. Debug WIRE, for example, uses bidirectional signaling on the RESET pin.

List of debuggers:

Some widely used debuggers are

- Allinea DDT
- GNU Debugger (GDB)
- Intel Debugger (IDB)
- LLDB
- Microsoft Visual Studio Debugger
- Valgrind
- WinDbg

There are two debugging tools. The first is a trace mechanism that can be used on the global functions in the top level loop. The second tool is a debugger that is not used in the normal top level loop. After a first program run it is possible to go back to breakpoints, and to inspect values or to restart certain functions with different arguments. This second tool only runs under Unix, because it duplicates the running process via a fork .

- ***Trace:***

The trace of a function is the list of the values of its parameters together with its result in the course of a program run.

The trace commands are directives in the toplevel loop. They allow to trace a function, stop its trace or to stop all active traces. These three directives are shown in the table below.

#trace name	trace function name
#untrace name	stop tracing function name
#untrace_all	stop all traces

Here is a first example of the definition of a function f:

```
# let f x = x + 1;;
val f : int -> int = <fun>
# f 4;;
- : int = 5
```

Now we will trace this function, so that its arguments and its return value will be shown.

```
#  #trace f;;
f is now traced.
# f 4;;
f <-- 4
f --> 5
- : int = 5
```

Passing of the argument *4* to f is shown, then the function f calculates the desired value and the result is returned and also shown. The arguments of a function call are indicated by a left arrow and the return value by an arrow to the right.

- ***Functions of Several Arguments:***

Functions of several arguments (or functions returning a closure) are also traceable. Each argument passed is shown. To distinguish the different closures, the number of arguments already passed to the closures is marked with a *. Let the function verif_div take 4 numbers (a, b, q, r) corresponding to the integer division: $a = bq + r$.

```
# let verif_div a b q r =
   a = b*q + r;;
val verif_div : int -> int -> int -> int -> bool = <fun>
# verif_div 11 5 2 1;;
- : bool = true
```

Its trace shows the passing of 4 arguments:

```
#  #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

- **Debug:**

ocamldebug, is a debugger in the usual sense of the word. It permits step-by-step execution, the insertion of breakpoints and the inspection and modification of values in the environment.

Single-stepping a program presupposes the knowledge of what comprises a program step. In imperative programming this is an easy enough notion: a step corresponds (more or less) to a single instruction of the language. But this definition does not make much sense in functional

programming; one instead speaks of program events. These are applications, entries to functions, pattern matching, a conditional, a loop, an element of a sequence, etc.

Warning

This tool only runs under Unix.

Compiling with Debugging Mode

The -g compiler option produces a .cmo file that allows the generation of the necessary instructions for debugging. Only the bytecode compiler knows about this option. It is necessary to set this option during compilation of the files encompassing an application. Once the executable is produced, execution in debug mode can be accomplished with the following ocamldebug command:

ocamldebug [*options*] executable [*arguments*]

Take the following example file fact.ml which calculates the factorial function:

```
let fact n =
  let rec fact_aux p q n =
    if n = 0 then p
    else fact_aux (p+q) p (n-1)
  in
fact_aux 1 1 n;;
```

The main program in the file main.ml goes off on a long recursion after the call of Fact.fact on -1.

```
let x = ref 4;;
let go () =
  x :=  -1;
  Fact.fact !x;;
go();;
```

The two files are compiled with the -g option:

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml
val fact : int -> int
val x : int ref
val go : unit -> int
```

Starting the Debugger

Once an executable is compiled with debug mode, it can be run in this mode.

```
$ ocamldebug fact.exe
        Objective Caml Debugger version 3.00

(ocd)
```

Execution Control

Execution control is done via program events. It is possible to go forward and backwards by *n* program events, or to go forward or backwards to the next breakpoint (or the nth breakpoint). A breakpoint can be set on a function or a program event. The choice of language element is shown by line and column number or the number of characters. This locality may be relative to a module.

In the example below, a breakpoint is set at the fourth line of module Main:

```
(ocd) step 0
Loading program... done.
Time : 0
Beginning of program.
(ocd)  break @ Main 4
Breakpoint 1 at 5028 : file Main, line 4 column 3
```

The initialisations of the module are done before the actual program. This is the reason the breakpoint at line 4 occurs only after 5028 instructions.

We go forward or backwards in the execution either by program elements or by breakpoints. run and reverse run the program just to the next breakpoint. The first in the direction of program execution, the second in the backwards direction. The step command advanced by 1 or *n* program elements, entering into functions, next steps over them. backstep and previous respectively do the same in the backwards direction. finish finally completes the current

functions invocations, whereas start returns to the program element before the function invocation.

To continue our example, we go forward to the breakpoint and then execute three program instructions:

```
(ocd) run
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) step
Time : 7 - pc : 4860 - module Fact
2   <|b|>let rec fact_aux p q n =
(ocd) step
Time : 8 - pc : 4876 - module Fact
6 <|b|>fact_aux 1 1 n;;
(ocd) step
Time : 9 - pc : 4788 - module Fact
3     <|b|>if n = 0 then p
```

Inspection of Values

At a breakpoint, the values of variables in the activation record can be inspected. The print and display commands output the values associated with a variable according to the different depths.

We will print the value of n, then go back three steps to print the contents of x:

```
(ocd) print n
n : int = -1
(ocd) backstep 3
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) print x
x : int ref = {contents=-1}
```

Access to the fields of a record or via the index of an array is accepted by the printing commands.

```
(ocd) print x.contents
1 : int = -1
```

Execution Stack

The execution stack permits a visualization of the entanglement of function invocations. The backtrace or bt command shows the stack of calls. The up and down commands select the next or preceding activation record. Finally, the frame command gives a description of the current record.

FAQs.

1. What is the Debugger?
2. Exlain any one Debugging tool in detail?
3. What is difference between Trace and Debug?