Savitribai Phule Pune University

System Programming & Operating System

T. Y. B. Sc. (Computer Science)

CS -347 SEMESTER III

Name _____

College Name _____

Roll No. _____ Division _____

Academic Year _____

Prepared By

Dr. Shailaja C. Shirwaikar

Prof. Nitin Patil

Reviewed By

Prof. Manisha Bharambe

Prof. Shubhangi Page

Prof. Jeevan Limaye

Preface

System programming and Operating systems course is as important component of any computer related course syllabi as much as System programs and Operating system are important to any computing system. The lab work designed for this course not only enhances the understanding of the subject but is a great programming experience. Writing simple toy level system programs and operating system components gives student a first hand experience in developing utility programs from scratch. This Lab Book supplements the text books and classroom teaching of System Programming and Operating System. The intention is to bring uniformity in conducting the lab sessions across various affiliated colleges. The assignments are designed so that the theory concepts in the syllabus are broadly covered. There is scope for improvement and additions and deletions can be carried out as the Lab book is always going to remain in digital form and available on the Department of Computer Science, Savitribai Phule Pune, website. I am indebted to all the reviewers of the book as their valuable suggestions have improved the book contents. We are all indebted to Dr. Vilas Kharat, Chairman, Board of studies in Computer Science for continuous encouragement, support and guidance.

Dr. Shailaja C. Shirwaikar

Member, Board Of Studies, Computer Science

Savitribai Phule Pune University

Table of contents

Introduction

1. About the work book

This workbook is intended to be used by T. Y. B. Sc (Computer Science) students for the Lab course in System Programming course in semester III and Operating system course in Semester IV.

System programming is the activity of designing and implementing System software. System Software consists of a variety of programs that assist in the use of a computer system. Operating System is a system software that takes the responsibility, on behalf of users, of managing and protecting the hardware. It provides an interface to the users so that their software programs can be executed easily and efficiently. Apart from Operating system, System software comprises of a large set of software mainly software processors and software tools.

Software processors such as editors, assemblers, Compilers etc and Operating system components such as shell, kernel etc. were some of the first software programs to get developed and their developers faced problem situations and came up with appropriate design strategies while implementing solutions to them. This course is intended to give an hands on experience on understanding these design principles, choosing appropriate data structures and choosing appropriate control structures for implementing wide range of algorithms.

This development experience will not only make you understand system programming and Operating system concepts but will equip you with design and implementation tricks that you will be able to use when you design your own software systems.

The objectives of this book are

1) Defining clearly the scope of the course
2) Bringing uniformity in the way the course is conducted across different colleges
3) Continuous assessment of the course
4) Bring in variation and variety in the experiments carried out by different students in a batch
5) Providing ready reference for students while working in the lab
6) Catering to the need of slow paced as well as fast paced learners

2. How to use this workbook

The workbook is divided into two sections. Section I is related to assignments in System Programming and Section II relates to assignments in Operating System. Both are to be implemented in C programming language in Linux environment. Printouts of completed assignments are not mandatory.

2.1 Instructions to the students

Please read the following instructions carefully and follow them.

1) Students are expected to carry this book every time they come to the lab for computer science practicals.

2) Students should prepare oneself before hand for the Assignment by reading the relevant material.

3) Instructor specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.

4) Students will be assessed for each exercise on a scale from 0 to 5

i) Not done	0
ii) Incomplete	1
iii) Late Complete	2
iv) Needs improvement	3
v) Complete	4
vi) Well Done	5

2.2. Instruction to the Instructors

1) Explain the assignment and related concepts in around ten minutes using white board if required or by demonstrating the software.

2) Make available to students digital copies of text files provided with the book as per the requirement of Assignment,

3) Make sure that students follow the instruction as given above.

4) You should evaluate each slot of assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.

5) The value should also be entered on assignment completion page of the respective Lab course.

2.3. Instructions to the Lab administrator

You have to ensure appropriate hardware and software is made available to each student.

The operating system and software requirements on server side and also client side are as given below:

1) Server and Client Side - (Operating System) Fedora Core Linux

2) Server side and Client Side - editor and GCC compiler

Assignment Completion Sheet

Lab Course I				
Section I – System Programming				
Sr. No	**Assignment Name**		**Marks (out of 5)**	**Signature**
1	Line Editor	Slot 1		
		Slot 2		
2	SMAC0 Simulator	Slot 1		
		Slot 2		
		Slot 3		
3	Assembler	Slot 1		
		Slot 2		
		Slot 3		
4	Macro Processor	Slot 1		
		Slot 2		
		Slot 3		
5	DFA Driver	Slot 1		
Total (out of 60)				
Total (Out of 10)				
6	Development Utilities			

Assignment 1 : Line Editor

Software Description – Editors are used to create digital copies of source program. The main functions supported by an editor is editing, viewing and navigating through the text. A line editor limits all operations to a line of text. The line is indicated positionally by giving line number i.e its serial number in the text or contextually by specifying a context which uniquely identifies the position.

The file to be edited is taken as command line argument. An empty file is opened for editing if no argument is supplied.

The editor has two modes

In command mode it displays '?' as prompt and accepts single-line commands. If 'i' for insert or 'a' for append command is given, it goes into input mode and accepts lines as text. When a line containing a single '.' is given it goes back to command mode.

The program at the start displays 'lines :' followed by number of lines(0 if file is empty or not specified) and goes into command mode.

The Command format is a single character indicating the action followed by three optional integers separated by spaces. The character and intended actions are given in the table 1.

The second parameter n1 and the third parameter n2 specify the range of lines and the command is valid if 1 <= n1 <= n2 <= total lines in the file being edited. The default value for n1 is 1 and the default value for n2 is n1. For example a command ' p 3 4 ' will print lines starting from line no 3 to line no 4, a command ' p 3' will print line 3 just a ' p' command will print the first line. If n1 or n2 is greater than the total lines in the file then n1 or n2 is set to total lines in the file so that command 'p 1 1000' will print the file till the end if total lines in file are < 1000.

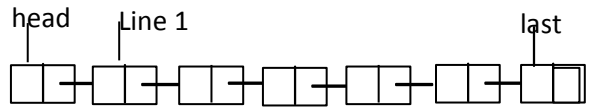
Command Character	Intended action
p	print or display
i	Insert
a	Append
d	Delete
m	Move
c	Copy
f	Find
s	Save
h	Help
q	Quit

Table 1: character and corresponding Action

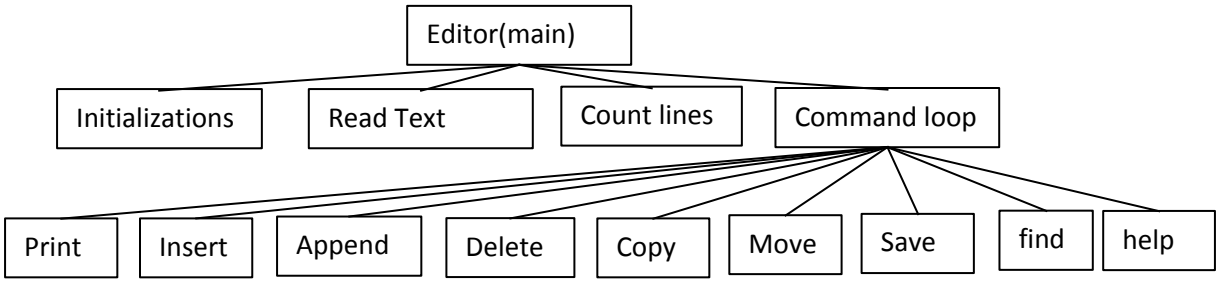
The second parameter and fourth parameter depending on the command, indicate the position of action. For example the command 'i 5' indicates that the lines entered are to be inserted from the 5th position that is fifth line onwards while the command 'm 2 4 5' indicate that the lines ranging from 2 to 4 should be moved to the 5th position.

Data Structure Design - Linked list of lines is the appropriate data structure for edit buffer that hold the lines to be edited, as lines are to be inserted, deleted, moved or copied. A singly linked list with a dummy header node can be used so that insertion deletion becomes easy.

Data Structure	'C' code

	typedef struct editbuff
	{
	char *line; // character string
	struct editbuff *next; // pointer to next line
	} List * head, *last;

Control structure – The design is modular. The main module performs necessary initializations, optionally reads text from file and starts a command loop which will process each of the possible commands. The structure diagram is as follows.



Procedural Design – The following table explains the input, algorithm and provides some implementation hints

Procedure	Description	Programming Hints
Editor(Main)	Input – optionally the filename	main(int argc, char *argv[])
	Perform initializations Read from the file optionally Count and display the line count Start the command loop	if(argc >1) { readbuff(head, argv[1]); printf(" Lines : %d", lines(head));

Initializations	Initializing the linked list	head = (List *) malloc (sizeof (List *));
		head->next=NULL;
		last=head;

Read text(file)	Input - list header and file name	void readbuff(List *head, char *filename)

	Open the file in read mode
	Initially last points to head
	while file not the end of file
	read a line from the file
	create a new node by allocating memory
	Allocate memory to string
	copy the line
	Attach the newnode to last
	Let last point to newnode
		If((fp=fopen(filename, "r+"))!=NULL)
		{ last =head;
		while(!feof(fp)) {
		if(!fgets(str,80,fp))break;
		temp = ….
		temp->line=(char malloc(strlen(str));
		strcpy(temp->line,str);
		last->next=temp;
		last=temp; }

Count line	Input – linked list header Output – no of lines	int lines(List *head)

	Start from the first node Traverse the list and increment the count	temp=head->next; while(temp !=NULL) { .... }

Command loop	Start the loop
	Prompt
	Read the command
	Separate the parameters
	
	Branch depending on the command character
	
	Exit the program on quit command
		while(1){
		printf("\n?");
		fgets(str,80,stdin);
		n=sscanf(str,"%c%d%d%d",
		&c,&n1,&n2,&n3);
		switch(c)
		{ case 'p':
		……..
		case 'q': exit(0);
		default : printf("wrong command");
		break;
		} }

Print	Input - list and two integer parameters n1,n2	void eprint(List *head, int n1, int n2)

	validate n1 , n2 and set default values
	Skip n1-1 lines
	
	Print lines from n1 to n2
	Line no : followed by line
		
		for(line=1, temp=head->next ; line<n1; line++) temp=temp->next;
		for(…) { printf("\nlineno %d :%s" ,line, temp->line)

Insert	Input - list and integer parameter n1 indicating position	

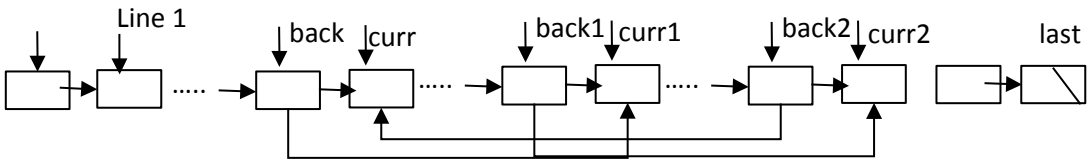
	Skip n1 lines using two pointers back following the current
	Start a loop
	Read a line
	Break if it consists of a single dot
	Store the line in a newnode
	Attach it next to back
	Let back point to new node
		for(..){ back=curr; curr=curr->next;}
		
		if (!strcmp(str, ".\n")) break;

Append	Input – list	
	Move to the last line using two pointers back following current Attach lines till one with a single dot	
Delete	Input – list and integer parameters n1 n2	
	Validate n1, n2 and set default values Skip n1 -1 lines using back and curr Skip n2 lines using back1 and curr1 Attach curr1 to back	 back ->next =curr1
Move	Input – list and integers n1, n2 and n3	

	Validate the parameters
	Skip n1-1 line with two pointers back and curr, skip n2 lines with two pointers back1 and curr1 and skip n3 lines with two pointers back2 and curr2
	Modify the pointers
	See Fig 1
		
		back2->next=curr;
		back1->next=curr2;
		back->next=curr1;

Copy	Input – list and integers n1, n2 and n3	
	Validate the parameters Skip n1-1 line with two pointers back and curr, skip n3 lines with two pointers back1 and curr1 Copy n2 lines Starting from curr each line in a new node and attach after back1	
Save	Input – list	

	Prompt “Filename :” to accept filename if it is empty If given file name is already exists check whether to overwrite this existing file contents (Yes/ No). If Yes then Open the file in write mode Put every line from the list into the file	if((fp=fopen(name, "w"))!=NULL) { while(temp != NULL) { fputs(temp->line,fp); ……}
Find	Input – list and range given by n1 and n2	
	Prompt “Pattern : “ to accept the pattern Skip n1=1 lines Traverse up to n2 lines Print the line If it contains the pattern	
Help	No parameters Display the list of commands with syntax description and examples	



Slot 1

i) Answer the following questions after carefully reading the description and program structure.

a) How main function is declared in case of command line program?
_____

b) What are the two modes of the editor?
_____

c) What data structure is appropriate for the edit buffer? Why?
_____

d) What command will print all lines in the file?
_____

e) What command will print nth line? What will print last line?
_____

f) What will command ‘d 5’ will do? What effect ‘d’ command will have?
_____

ii) Partially implement a command line program for a line editor. Implement the following functionalities

a) The program accepts the filename and prints the number of lines in the file and prompts for the command

b) Implement the print command

c) Implement the insert command

d) Implement the save command

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 2

i) Extend the line editor

a) Implement the delete command

b) Implement the move command

c) Implement the copy command

d) Implement the find command

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 3 (Optional)

i) Implement the help command

ii) Change the data structure to a doubly linked list. The print command 'p m n ' is now valid if m > n and also if m < n, wherein it prints lines backwards from m to n. Modify the print command accordingly

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Assignment 2 : SMAC0 simulator

Software Description – A simple instruction Computer(SMAC0) is a hypothetical machine with a small but effective instruction set that can be used to illustrate the design of simple software processors involved in development of programs such as Assembler, Macro processor etc. The machine will incorporate the most commonly encountered hardware features and concepts, while avoiding irrelevant complexities.

A simulator program is required that simulates the function of simple instruction computer such as fetching an instruction, decoding and executing it.

The hypothetical Simple machine (SMAC0) has following features.

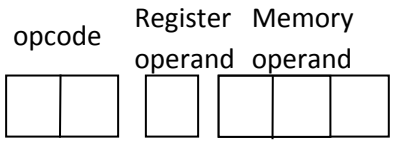
Memory – Memory consist of 6 digit words (decimal). Total size of memory is 1000 words (10^3) indicates the address size is 3 digits (address ranges from 0 to 999).

Registers – There are in all six registers four general purpose registers AREG, BREG, CREG and DREG numbered 1,2,3 and 4.

A program counter (PC) storing the address of the next instruction to be fetched and a status register storing condition codes. There are SIX condition codes LT, LE, EQ, GT , GE and ANY numbered 0,1,2,3,4 and 5. Each bit in the status register can be set to 1

Data Format – Supports only six digit integer data stored in decimal form.

Instruction Format – It has single instruction format. Each instruction is of six digit length. The opcode, register operand and memory operand occupy 2, 1 and 3 digits in that order



Instruction Set

Opcode	Mnemonic	Instruction	Operands
00	STOP	Stop or Halt execution	Operands unused
01	ADD	Add memory operand to register operand	Register and memory operand
02	SUB	Subtract memory operand from register operand	Register and memory operand
03	MULT	Multiply memory operand to register operand	Register and memory operand
08	DIV	Divide register operand by memory operand	Register and memory operand
04	MOVER	Move memory operand contents to register operand	Register and memory operand

05	MOVEM	Move register operand contents to memory	Register and memory operand
06	COMP	Compare register and memory operands to set condition code appropriately	Register and memory operand
07	BC	Branch to second operand depending on condition code specified as first operand	Register and memory operand
09	READ	Read into memory operand	Only memory operand
10	PRINT	Print contents of memory Operand	Only memory operand

Condition code	Mnemonic	Description
0	LT	Less than
1	LE	Less than or equal to
2	EQ	Equal to
3	GT	Greater than
4	GE	Greater than or equal to
5	ANY	Unconditional

It should be possible to load program from file into memory at specified location. File contains program as sequence of lines, each line containing address followed by content indicating the instruction to be stored at that address. The file ends with –1 followed by starting address indicating physical end of file.

Simple program to add two numbers		
address	Content	Description
100	090107	Read into 107th memory address
101	090108	Read into 108th memory address
102	041107	Move contents of 107th memory address to register 1
103	011108	Add contents of 108th memory address to register 1
104	051109	Move contents of register 1 to memory address 109
105	100109	Output contents of 109th memory address
106	000000	Halt – logical end of the program
107	0	Address to be used for first integer
108	0	Address to be used for second integer
109	0	Address to be used for result

The above program should be stored in a file sum.sm as follows

```
100 90107
101 90108
102 41107
103 11108
104 51109
105 100109
106  0
-1    100
```

Similarily the program for printing factorial of the number read is given below. Store it in a file fact.sm.

```
100 090113
101 041113
102 042112
103 061112
104 071109
105 032113
106 021112
107 051113
108 075103
109 052114
110 100114
111 0
112 1
-1 100
```

The simulator program should be menu driven supporting.

Load – loading the program into memory from the file after accepting filename.

Print – print the content of loaded program

Accept – accept the program as string of address content pairs

Run – execute the program

Trace – execute statement by statement displaying contents of all the registers

Quit – quits the program

The menu should look like

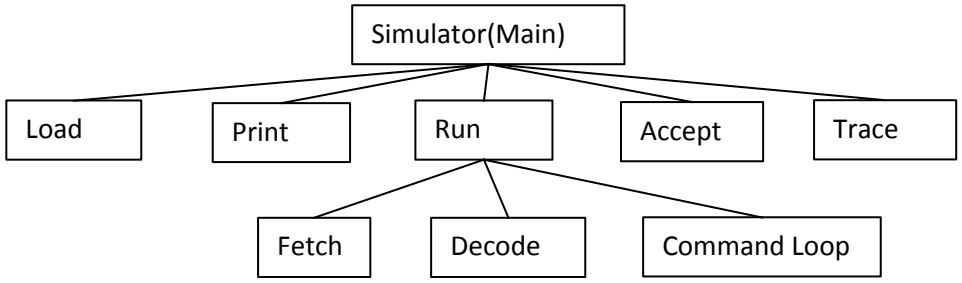
```
1:    Load
2:    Print
3:    Accept
4:    Run
5:    Trace
6:    Quit
Choose option by specifying corresponding integer
```

Data Structure Design – The SMAC0 machine has memory and a set of registers. Appropriate data structures need to be chosen to represent each one of themor. Simulator program also need to store the last valid address in the physical file.

Component	Description	'C' code
Memory	An array of 1000 words each can store an integer	int mem[1000];

Program Counter	An integer indicating the address of instruction	int pc;
General Purpose registers	Four general purpose registers numbered 1, 2, 3 and 4	int reg[4];
Condition Code register	A single integer with each bit representing a condition code or An array of six registers each storing condition code separately	int cc; int cc[6];
Last address	An integer indicating last valid address in physical file	int lc;

Control structure – The design is modular. The main module provides the menu options and allows one to choose the appropriate option.



Procedural Design – The following table explains the input , algorithm and provides some implementation hints

Procedure	Description	Programming Hints

Simulator(main)	In a loop
	Print options
	Read an option
	Branch depending on option
		do {
		printoptions();
		scanf("%d",&option);
		switch(option){case 1: ……
		} while(option!=6)

Load	Accept the filename
	Open the file for reading
	Read from the file
		
		fscanf(fp, "%d%d", &address, &content);
		if (address==-1) pc= content;
		else { lc = address; mem[address]=content;

Print	Print from contents from pc to lc	for(i=pc ; i<=lc; i++)
Run	Input – start address	void execute(int pc)

	Fetch the instruction
	Decode the instruction
	
	While opcode is not zero
	Depending on opcode take action
	If opcode is 1 add memory operand to register
	…
	If opcode is 4 move contents of memory operand to register operand
	…..
	If opcode is 6 compare mem operand with register operand and set the condition code
	
	If opcode is 7 take the jump to memory operand if condition code matches
	If opcode is 9 read into memory operand ……
		mem[pc] // contains the instruction
		opcode=mem[pc]/10000;
		// separate register and memory operand
		
		Switch(opcde){
		case 1: reg[regop]+=mem[memop];pc++;break;
		
		case 4 : reg[regop]=mem[operand];pc++;break;
		
		case 6: for(i=0;i<6;i++) cc[i]=0;
		if (reg[regop]< mem[operand]) cc[0]=1;
		…
		if (reg[regop]>= mem[operand]) cc[4]=1;
		pc++; break;
		
		if((cc[regop]==1)||(regop==5))
		pc=operand; else pc++; break;
		
		printf(" give the contents");
		scanf("%d", &mem[operand]); pc++; break;

Accept	Accept address and content till address equals -1 and store them in memory Optionally allow saving them in a file	
Trace	Set the trace flag Call modified Run function that outputs contents of all registers when the trace flag is on	

Slot 1

i) Answer the following questions after carefully reading the description and program structure.

a) What is the size of memory in Hypothetical Simple Instruction Computer(SMAC0)? How memory will be represented in simulation program?

_____

_____

b) How many registers are there in SMAC0? How registers are represented in simulation program?

_____

_____

c) From the contents of the memory at pc, how will you separate opcode, register operand and memory operand?

ii) Implement a menu driven simulator for hypothetical Simple Instruction Computer that provides the following functionalities

a) Load - Loading of the program from file into memory

b) Print - Printing the program loaded in memory

c) Run - Executing the loaded program

The machine has the basic instruction set comprising of add, mover, movem, read, print and hlt commands as given in Table 1. Create a file sum.sm containing the machine code for sum of two numbers. Test the program using the machine code programs sum.sm.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 2

i) Extend the program by adding the following functionalities

a) Accepting of the program from the user and storing it in file

b) Trace option that executes the program statement by statement displaying the contents of the registers

c) Extend the instruction set to include sub, mult, div, comp and bc instruction

ii) Create a file fact.sm containing the machine code for printing factorial of number read.Test the program using the machine code program fact.sm.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 3 i) Extend the instruction set further to include the following

Opcode	Mnemonic	Instruction	Operands
11	SWAP	Swap the contents of memory and register operand	Both register and memory operand
12	INCR	Increment the contents of register operand	Only register operand
13	DECR	Decrement the contents of register operand	Only register operand
14	INCM	Increment the contents of memory operand	Only memory operand
15	DECM	Decrement the contents of memory operand	Only memory operand
16	ADDM	Add the contents of register operand to memory operand	Both the operands
17	SUBM	Subtract the contents of register operand from memory operand	Both the operands
18	MULTM	Multiply the contents of memory operand by register operand	Both the operands
19	DIVM	Divide the contents of memory operand by register operand	Both the operands
20	PRINTR	Print the contents of register operand	Only register operand
21	READR	Read into the register operand	Only register operand
22	ZEROR	Initialize register operand to zero	Only register operand
23	ONER	Initialize register Operand to One	Only register operand

ii) Test after converting the following programs to machine code

Sum of two numbers	Maximum of two numbers	Factorial of number
READR AREG READ A ADD AREG A PRINTR AREG STOP	READR AREG READ A COMP AREG A BC GE NEXT SWAP AREG A NEXT PRINTR AREG STOP	ONER BREG ONER AREG READ A AGAIN COMP AREG A BC GE OUT MULT BREG A DECM A BC ANY AGAIN OUT PRINTR BREG STOP

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion ____/____/_____**

Assignment 3 : Assembler

Software Description – An Assembler is a software processor that takes as input an assembly language program and translates it into machine code if it is error free otherwise provides a list of errors. We will design an assembler for the hypothetical simple Instruction computer and its assembly language.

Apart from imperative statements an assembly language contains Assembler directives and declaration statements. The assembler should be able to handle programs containing assembler directives 'START' and 'END', declaration statements 'DS' and 'DC' and the imperative statements.

The Assembler will have two passes. In the first pass it will generate intermediate code and create symbol table. In the second pass intermediate code will be converted to machine code

Data Structure Design – A design of assembler requires several tables such as symbol table, mnemonic table, intermediate code table and error table. Each symbol encountered in source program is added to symbol table Each symbol table entry stores symbol name, address and two flags indicating whether symbol has been used and whether it is defined. When Symbol appears as label, it gets defined and corresponding address gets added to the table. It is used when it appears as an operand. There are three mnemonic tables one for opcodes, the other for general purpose registers and the third for the condition codes. The intermediate code table stores intermediate code for each source line. Each table entry stores address, opcode, register operand number, character which can be 'S' or 'C' indicating Symbol or constant and the value which is index of symbol table entry or actual value in case of constant. The error table stores line number and type of error indicated by error number while error message table contains error messages for each error number. Apart from tables, the assembler uses a pointer 'lc' to the current line being processed. The implements of tables can be static or dynamic.

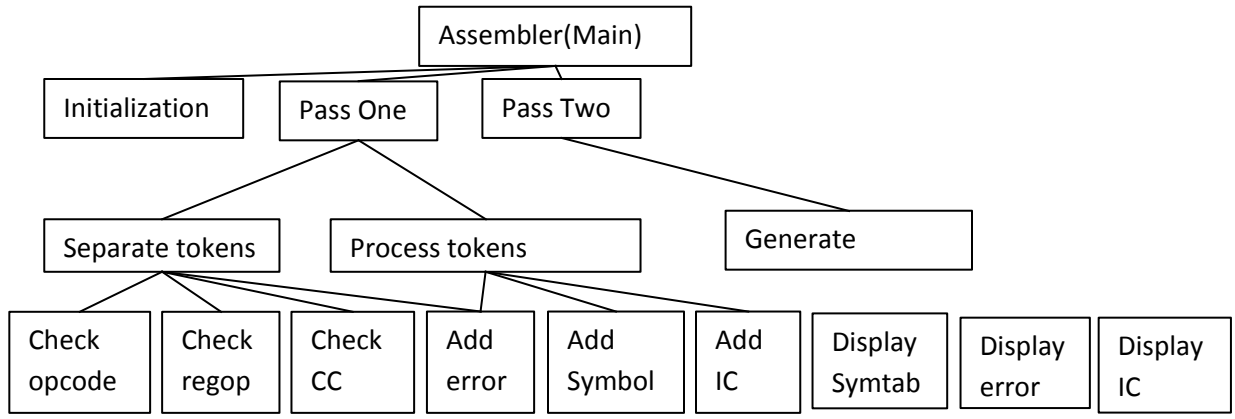
Component	Description	'C' code
Mnemonic table	Table that stores valid mnemonics and also the index matches the opcode or imperative statements	*mne[15]={"STOP","ADD",……"DC","START","END"}
Register table	Table stores register names and index indicates the register number	char *reg[4]={"AREG", "BREG","CREG","DREG"};
Condition Code table	Table stores condition names and index indicates the condition code	char *cc[6]={"LT","LE","EQ","GT","GE", "ANY"};

Symbol table	Each entry in symbol table contains the name, address and flags indicating whether the symbol is used and defined	struct symtab { char symbol[20]; int add; int used; int defined; }sym[50];
Intermediate code table	Each table entry stores address, opcode, register operand number, character which can be 'S' or 'C' indicating Symbol or constant and the value	struct ictab{ int address; int opcode; int regop; char optype; int value; }ic[50];
Error table	Each entry contains line number and the error number indicating type of error	struct errtab{ int lineno; int errno; }err[50];
Error message table	This table is used for giving different error messages	char *errmsg[6]={"used but not defined", "invalid opcode","wrong statement format",..};

Control structure – The assembler is two pass so the main module calls Pass1 followed by Pass2 if there are no errors. The file to be assembled is provided as command line argument.in pass one, every line in source program is separated into label, opcode, register and memory operand. In some cases label is empty while in some cases the operands. Each token is validated so also the statement format and appropriate error is added to error table. The separated tokens are then processed

If label is present it is added to symbol table with appropriate attribute values so also the memory operands. The intermediate code generated for every line is added to Intermediate

code table.



Procedural Design – The following table explains the input , algorithm and implementation hints for some of the procedures

Procedure	Description	Programming Hints
Assembler(main)	Input – source file as command line argument	
	Apply Passone to source program If no errors then apply Passtwo	If(argc==2) passone(argv[1]);
Passone	Open the file In a loop Read a source program line Separate tokens Process tokens	
Passtwo	In a loop Get an entry from IC Refer symbol table	

Separate tokens	Split the input string into strings
	Check if number of tokens are 4
	If mnemonic is valid
	If opcode requires two operands
	check validity of condition code
	check validity of register operand
	copy label, opcode and operands
	if invalid add error to error table
	Check if number of tokens are 3
	Check if mnemonic requires two operands, validate and copy
	Check if number of tokens are 2
	handle different possibilities
	Check if number of tokens are 1
	It can be END or STOP
		n=sscanf(str,"%s%s%s%s", s1,s2,s3,s4);
		if(n==4)
		{ if((c=checkm(s2))!=-1)
		if(c>=1 && c <=8)
		
		adderror(lno,3);}
		if(n==3)
		
		{if(c >=9 && c <=13)
		If (n==2) {
		
		if(n==1) {

Process tokens	if label is present
	add label to symbol table as defined along with lc
	if mnemonic is start
	change lc
	if mnemonic is DS
	modify symbol table and change lc
	if mnemonic is DC
	add appropriate entry IC table
	if imperative opcode
	add operand2 to symbol table as used if not present
	add appropriate entry to IC table
		if(!strcmp(label,""))
		
		`if ( opcode==12) lc+=atoi(op2);`
		if (opcode ==13)

Check opcode	Check if the mnemonic is present and return the index in the table as opcode else return -1	int checkm(char * str)
		{ int i;
		for(i=0; i<15; i++)
		if(!strcmp(str,mne[i])) return i;
		return -1;
		}

Add Sym	Check if symbol is present and update or add the symbol to symbol table	
Display Error table	Display logged errors and corresponding error messages	

Slot 1 i) Answer the following questions after carefully reading the description and program structure.

a) What data structures are used by the first pass of assembler?

_____

b) How mnemonic table will be implemented in C?

_____

c) Give the declaration for Symbol table.

_____

_____

_____

ii) Implement a Two pass Assembler for hypothetical simple Instruction Computer and its simple assembly language that includes Assembler directives ”START” and “END”, the declarative statements “DS” and “DC” and imperative statements with mnemonics “STOP” to “ “PRINT”

a) Implement necessary tables statically and write functions for checking, displaying adding to tables.

b) Store test program given below in a file and write dummy Passone that only prints the source program lines with line nos

```
START 300
BEGIN READ  NUM
LOOP MOVEM AREG NUM
PRINT NUM
MULT AREG NUM
COMP AREG HUNDRED
BC LT LOOP
STOP
NUM DS 2
HUNDRED DC ‘100’
END BEGIN
```

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 2

i) Implement Separatetokens

ii) Verify using test program the separation of tokens for each line

iii) Display the error table

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 3

i) Implement process tokens

ii) Display the contents of symbol table, error table and IC table

iii) Implement Pass2

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 4 (Optional)

i) Implement tables using dynamic data structure and modify code accoordingly.

ii) Add ORIGIN as 16th mnemonic and make appropriate changes to separate tokens and processtokens

iii) Add EQU as 17th mnemonic and make appropriate changes to separate tokens and processtokens

iv) Define literal table as data structure.

In processtokens check If symbol is a literal (starts with = sign) then add it to literal table

Write a functions for processing literals and call it at the end of passone

Change passtwo accordingly

v) Add LTORG as 18th mnemonic and make appropriate changes to code

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Assignment 4 : Macro Processor

Software Description – An assembly language macro is a facility for extending the set of operations provided in an assembly language. A programmer can define his own set of macros only once and can use them many times.

A macro definition consists of a name , a set of formal parameters and a body of code. When a macro name along with a set of actual parameters is used, it is replaced by body of macro and it is called macro expansion.

Macro processor is a software that takes as input a program containing macro definitions and calls and generates an assembly language program which is free of macro definitions and where macro calls have been properly expanded. Macro processor has two main steps

i) Processing macro definitions ii) macro expansion

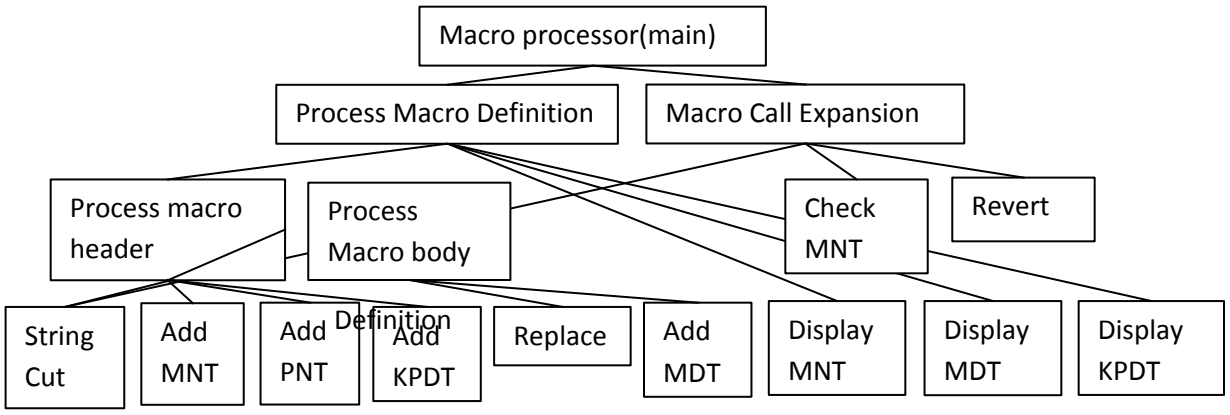
In the first step each macro definition is processed to extract information and is stored in well defined data structures. In macro expansion each macro call is expanded using appropriate information from the tables.

Data Structure Design – The design of macro processor requires several tables. The first preprocessing step uses tables such as macro name table, Keyword parameter default value table, macro definition table, parameter name table. Macro expansion step uses actual parameter name table, macro name table, Keyword parameter default value table and macro definition table. Since additions are to be done to all these tables we need pointers indicating last vacant position in the table.

Component	Description	'C' code
Macro name table	It stores the name of the macro and other information such as no of positional parameters, keyword parameters etc. It is used as a lookup table when a macro call is identified.It also contains pointers to all other tables where relevant information is stored	struct mnttab{ // structure of MNT table char name[30]; int pp; int kp; int kpdptr; int mdtptr; }mnt[10];
Parameter name table	It contains names of formal parameters including positional and keyword parameters	char pnt[10][30]; or char **pnt;
keyword parameter default value table	It contains keyword parameters and their default values	
Macro definition Table	It contains the model statements of all macros . They are kept in partially processed (IC) form so that expansion is easier	

Actual parameter Table	It contains actual parameters i.e. values that will replace formal parameters during the expansion	
Pointers to various tables		int mdtptr=0; ….

Control structure – The macro processor in first step stores extracted information in tables when macro definitions are encountered and in second step performs expansion after validating each macro call.



Procedural Design – The following table explains the input, algorithm and implementation hints for some of the procedures

Procedure	Description	Programming Hints
Macro Processo(main)	Input – source file as command line argument	

	Open the file and read line by line
	For each macro definition
	Extract information from macro header statement and add to tables
	Store macro definitions statements in MDT table till mend statement is reached after replacing parameters by positional markers.
	For each macro call extract name and actual parameters
	Expand after replacing macro definition statements corresponding to macro name by actual parameters
		if(!strcmp(str, "macro\n"))
		
		while (strcmp(str,"mend\n"))
		{ …
		replace(str, str1)
		}

Extract	Extract macro name Extract all positional parameters that start with & and add to PNT table Extract all keywords that start with = and add keyword parameters and default values to KPDT table Add name, parameter count and pointers to MNT table	s=strcut(s,mname);
Expand	Extract macro name from macro call Check if it is present in MNT table Get mdtptr and kpdptr from table Prepare actual parameter table and add default values Extract and add actual parameters Appropriately revert positions by parameters in statements in MDT table starting from mdtptr till MEND	
Replace	if parameter is present in macro body replace it by (P,n) where n is the parameter position	
Revert	Replace (P,n) in macro body by parameter name at nth position in actual parameter name table	

Slot 1 i) Answer the following questions after carefully reading the description and program structure.

a) What are the two main tasks of a macro processor?

_____

b) What tables are used by macro processor to store extracted information?

_____

_____

ii) Create a file named first.asm containing following macro definitions

```
MACRO
COPY &ONE, &TWO, &REG=BREG
MOVER &REG,  &ONE
MOVEM &REG, &TWO
MEND
MACRO
CHANGE &FIRST, &SECOND, &REG=AREG, &OP=ADD
MOVER &REG, &FIRST
&OP &REG, &SECOND
MOVEM &REG, &FIRST
MEND
```

iii) Write a command line macro processor program that takes above file as command line argument and prints the macro names, and names of parameters.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 2

i) Extend the macro processor program

a) Define appropriate data structure for all the tables

b) Write code for extracting information from

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 3

i) Extend first.asm by attaching the code below

```
READ A
COPY A, B
CHANGE A, B, REG=CREG
COPY A, C
CHANGE C, B , OP=SUB, REG=DREG
PRINT A
PRINT B
PRINT C
STOP
A DS 1
B DS 1
C DS 1
END
```

ii) Extend the macro processor program that also expands the macro calls appropriately to create the final assembly language program

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 4 (Optional)

i) Implement various tables using dynamic data structures and modify the code accordingly

Assignment Evaluation

0: Not Done [ ] 1: Incomplete [ ] 2: Late Complete [ ]

3: Needs Improvement [ ] 4: Complete [ ] 5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Assignment 5 : DFA Driver

Software Description – Finite automata is a mathematical model of a machine with finite number of internal configurations or states. Input to the machine is from a finite set of symbols that forms the alphabet denoted by ∑. Machine keeps on changing its state on consuming an input symbol and the state can be one among the finite set of states denoted by Q. These transitions can be specified either by giving a transition table or a transition diagram denoted by δ. The machine always starts in a specific state which is designated as start state and is denoted as q0. There are some states in Q which are final states or accepting states. The set of Final states is denoted by F. Thus a Finite automata is characterized by these five components and mathematically it is a five tuple {Q, ∑, δ, qo, F}.

The language accepted by FA is the set of all strings for which the FA halts in a final state. The languages accepted by FA are Regular languages. In case of Deterministic FA , the transitions are uniquely defined on a state and input symbol.

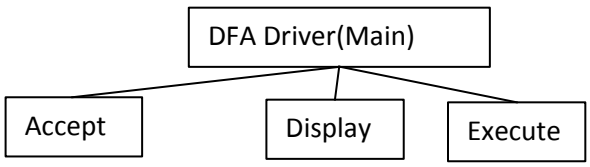
DFA driver is a software that helps to construct a DFA and execute it on a string.

Data Structure – DFA is a five tuple consisting of set of states, alphabet(set of symbols), transitions, start state and set of final states. We assume that states are numbered from 0 to n-1 and start state is always 0. The alphabet is always a, b, c in alphabetical order. The set of final states can be defined using an array of 0s and 1s , a 1 indicates that the state is final. Transitions are defined using transition table which is a two dimensional array. Thus DFA components are number of states, number of symbols, transition matrix and array of final states.

Component	Description	'C' code

DFA	DFA consists of number of states, number of symbol, transition table, start state is assumed by default as 0 and the Boolean array of final states	struct DFA {
		int m; // no of states
		int n; //no of symbols
		int delta[10][10]// transition table
		int final[10];// array of final states
		}

Control structure – The DFA driver accepts the DFA for a given language and then it can be executed on any string. The output is 'accepted' if string is in the language or 'rejected' if string is not in the language.



Procedure	Description	Programming Hints
DFA Driver (main)	Accept the DFA details from user or initialize DFA Accept the string Execute the DFA over the string and output whether string is accepted or rejected	
Accept	Accept the number of states, number of symbols Accept transition of every state over every alphabet Accept final states Or Initialize dfa	struct DFA odd={2,2,{ {1,1}, {0,0} },{0,1} };
Display	Display DFA as a five tuple with its transition table	
Execute	Initialize Current state to start state For every symbol in the input string current state is transformed to transition from current state over the input symbol Output accepted if current state is final and rejected otherwise	

Slot 1

i) Answer the following questions after carefully reading the description and program structure.

a) How will you initialize the DFA for the language L={ the set of all strings over {a,b} that start with a }

_____

b) How will you initialize the DFA for the language L={ the set of all strings over {a,b, c} that contain substring ‘aa’ }

_____

c) How will you initialize the DFA for the language L= _____

_____

_____

d) How will you initialize the DFA for the language L= _____

_____

_____

ii) Implement a DFA driver that allows initializing a DFA, display and executes a DFA

iii) Extend DFA driver to accept DFA details from user.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____

Slot 2 (optional)

i) Implement a NFA driver that accepts an NFA, converts NFA to DFA and displays the corresponding DFA

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor **Date of Completion** ____/____/_____