



**NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE**  
(NAAC Accredited)



*(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)*

**Pampady, Thiruvilwamala (PO), Thrissur (DT), Kerala 680 588**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**LAB MANUAL**



***CS331 SYSTEM SOFTWARE LAB***

**VISION OF THE INSTITUTION**

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

**MISSION OF THE INSTITUTION**

**NCERC** is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

## **ABOUT THE DEPARTMENT**

- ◆ Established in: 2002
- ◆ Course offered: B.Tech. in Computer Science and Engineering  
M. Tech. in Computer Science and Engineering  
M. Tech. in Cyber Security
- ◆ Approved by AICTE New Delhi and Accredited by NAAC
- ◆ Certified by ISO 9001-2015
- ◆ Affiliated to A P J Abdul Kalam Technological University, Kerala.

## **DEPARTMENT VISSION**

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

## **DEPARTMENT MISSION**

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

## **PROGRAMME EDUCATIONAL OBJECTIVES**

- PEO1:** Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.
- PEO2:** Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.
- PEO3:** Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.
- PEO4:** Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Team work and leadership qualities.

## PROGRAM OUTCOMES (POs)

### Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

### **PROGRAM SPECIFIC OUTCOMES (PSO)**

**PSO1:** Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

**PSO2:** Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

**PSO3:** Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

### **COURSE OUTCOME**

<b>CO 1</b>	Design and implement programs on 8086 microprocessor
<b>CO 2</b>	To provide solid foundation on interfacing the external devices to the processor according to the user requirements
<b>CO 3</b>	Design and implement 8051 microcontroller based systems
<b>CO 4</b>	To Understand the concepts related to I/O and memory interfacing
<b>CO 5</b>	To learn about interfacing stepper motor working and its interfacing
<b>CO 6</b>	To learn about generation of waveforms using microcontroller
<b>CO 7</b>	To learn about different types of flag registers and their changes while performing arithmetic operations

**MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES**

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO 1	1	3	3	3	3	3						
CO 2	2	1	1	2	1			2	2	1	3	2
CO 3	1	1	2	2	3		3	3	3	3	3	3
CO 4	1							2	2	1	3	2
CO 5						3						
CO 6												3
CO 7	1	2	2	1	2	3	1	1		3		1

**MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES**

	PSO1	PSO2	PSO3
CO1	3	3	-
CO2	3	3	-
CO3	3	3	-
CO4	3	3	-
CO5	3	3	-
CO6	3	3	-

**Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1**

## **PREPARATION FOR THE LABORATORY SESSION**

### **GENERAL INSTRUCTIONS TO STUDENTS**

1. Read carefully and understand the description of the experiment in the lab manual. You may go to the lab at an earlier date to look at the experimental facility and understand it better. Consult the appropriate references to be completely familiar with the concepts and hardware.
2. Make sure that your observation for previous week experiment is evaluated by the faculty member and you have transferred all the contents to your record before entering to the lab/workshop.
3. At the beginning of the class, if the faculty or the instructor finds that a student is not adequately prepared, they will be marked as absent and not be allowed to perform the experiment.
4. Bring necessary material needed (writing materials, graphs, calculators, etc.) to perform the required preliminary analysis. It is a good idea to do sample calculations and as much of the analysis as possible during the session. Faculty help will be available. Errors in the procedure may thus be easily detected and rectified.
5. Please actively participate in class and don't hesitate to ask questions. Please utilize the teaching assistants fully. To encourage you to be prepared and to read the lab manual before coming to the laboratory, unannounced questions may be asked at any time during the lab.
6. Carelessness in personal conduct or in handling equipment may result in serious injury to the individual or the equipment. Do not run near moving machinery/equipment. Always be on the alert for strange sounds. Guard against entangling clothes in moving parts of machinery.
7. Students must follow the proper dress code inside the laboratory. To protect clothing from dirt, wear a lab coat. Long hair should be tied back. Shoes covering the whole foot will have to be worn.
8. In performing the experiments, please proceed carefully to minimize any water spills, especially on the electric circuits and wire.
9. Maintain silence, order and discipline inside the lab. Don't use cell phones inside the laboratory.

10. Any injury no matter how small must be reported to the instructor immediately.

11. Check with faculty members one week before the experiment to make sure that you have the handout for that experiment and all the apparatus.

#### AFTER THE LABORATORY SESSION

1. Clean up your work area.
2. Check with the technician before you leave.
3. Make sure you understand what kind of report is to be prepared and due submission of record is next lab class.
4. Do sample calculations and some preliminary work to verify that the experiment was successful

#### MAKE-UPS AND LATE WORK

Students must participate in all laboratory exercises as scheduled. They must obtain permission from the faculty member for absence, which would be granted only under justifiable circumstances. In such an event, a student must make arrangements for a make-up laboratory, which will be scheduled when the time is available after completing one cycle. Late submission will be awarded less mark for record and internals and zero in worst cases.

#### LABORATORY POLICIES

1. Food, beverages & mobile phones are not allowed in the laboratory at any time.
2. Do not sit or place anything on instrument benches.
3. Organizing laboratory experiments requires the help of laboratory technicians and staff. Be punctual.

**SYLLABUS**

Course code	Course Name	L-T-P Credits	Year of Introduction
CS331	SYSTEM SOFTWARE LAB	0-0-3-1	2016
<b>Prerequisite: Nil</b>			
<b>Course Objectives</b>			
<ul style="list-style-type: none"> <li>To build an understanding on design and implementation of different types of system software.</li> </ul>			
<b>List of Exercises/Experiments: (Exercises/experiments marked with * are mandatory from each part. Total 12 Exercises/experiments are mandatory)</b>			
<i>Part A</i>			
<ol style="list-style-type: none"> <li>Simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.               <ol style="list-style-type: none"> <li>FCFS</li> <li>SJF</li> <li>Round Robin (pre-emptive)</li> <li>Priority</li> </ol> </li> <li>Simulate the following file allocation strategies.               <ol style="list-style-type: none"> <li>Sequential</li> <li>Indexed</li> <li>Linked</li> </ol> </li> <li>Implement the different paging techniques of memory management.</li> <li>Simulate the following file organization techniques *               <ol style="list-style-type: none"> <li>Single level directory</li> <li>Two level directory</li> <li>Hierarchical</li> </ol> </li> <li>Implement the banker's algorithm for deadlock avoidance.*</li> <li>Simulate the following disk scheduling algorithms. *               <ol style="list-style-type: none"> <li>FCFS</li> <li>SCAN</li> <li>C-SCAN</li> </ol> </li> <li>Simulate the following page replacement algorithms               <ol style="list-style-type: none"> <li>FIFO</li> <li>LRU</li> <li>LFU</li> </ol> </li> <li>Implement the producer-consumer problem using semaphores. *</li> <li>Write a program to simulate the working of the dining philosopher's problem.*</li> </ol>			
<i>Part B</i>			
<ol style="list-style-type: none"> <li>Implement the symbol table functions: create, insert, modify, search, and display.</li> <li>Implement pass one of a two pass assembler. *</li> <li>Implement pass two of a two pass assembler. *</li> <li>Implement a single pass assembler. *</li> <li>Implement a two pass macro processor.*</li> <li>Implement a single pass macro processor.</li> <li>Implement an absolute loader.</li> <li>Implement a relocating loader.</li> <li>Implement pass one of a direct-linking loader.</li> <li>Implement pass two of a direct-linking loader.</li> <li>Implement a simple text editor with features like insertion / deletion of a character, word, and sentence.</li> <li>Implement a symbol table with suitable hashing.*</li> </ol>			



**Expected Outcome**

The students will be able to

- i. Compare and analyze CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.
- ii. Implement basic memory management schemes like paging.
- iii. Implement synchronization techniques using semaphores etc.
- iv. Implement banker's algorithm for deadlock avoidance.
- v. Implement memory management schemes and page replacement schemes and file allocation and organization techniques.
- vi. Implement system software such as loaders, assemblers and macro processor.

**INDEX**

<b>EXP NO</b>	<b>EXPERIMENT NAME</b>	<b>PAGE NO</b>
1	CPU SCHEDULING ALGORITHM.	11
2	FILE ORGANIZATION TECHNIQUES	15
3	BANKERS ALGORITHM	18
4	DISK SCHEDULING ALGORITHM.	21
5	PRODUCER-CONSUMER PROBLEM.	26
6	DINING PHILOSOPHERS' PROBLEM	31
7	PASS ONE OF A TWO PASS ASSEMBLER	34
8	PASS TWO OF A TWO PASS ASSEMBLER.	37
9	SINGLE PASS ASSEMBLER.	42
10	TWO PASS MACROPROCESSOR.	48
11	SINGLE PASS MACROPROCESSOR.	54
12	ABSOLUTE LOADER	60
13	RELOCATING LOADER	65
14	SYMBOL TABLE USING HASHING	70

## EXPERIMENT – 1

### CPU SCHEDULING ALGORITHMS

**AIM:** Simulate the following non-pre-emptive CPU scheduling algorithms to find turnaround time and waiting time.

- a) FCFS
- b) SJF
- c) Round Robin (pre-emptive)
- d) Priority

#### ALGORITHM

##### a) First Come First Serve (FCFS):

1. Jobs are executed on first come, first serve basis.
2. It is a non-pre-emptive, pre-emptive scheduling algorithm.
3. Easy to understand and implement.
4. Its implementation is based on FIFO queue.
5. Poor in performance as average wait time is high.

Wait time of each process is as follows –

Process Wait Time : Service Time - Arrival Time

$$P0 \ 0 - 0 = 0$$

$$P1 \ 5 - 1 = 4$$

$$P2 \ 8 - 2 = 6$$

$$P3 \ 16 - 3 = 13$$

$$\text{Average Wait Time: } (0+4+6+13) / 4 = 5.75$$

##### b) Shortest Job First (SJF)

1. This is also known as shortest job next, or SJN
2. This is a non-pre-emptive, pre-emptive scheduling algorithm.
3. Best approach to minimize waiting time.
4. Easy to implement in Batch systems where required CPU time is known in advance.
5. Impossible to implement in interactive systems where the required CPU time is not known.
6. The processor should know in advance how much time process will take.

Wait time of each process is as follows –

Process Wait Time : Service Time - Arrival Time

$$P0 \ 3 - 0 = 3$$

$$P1 \ 0 - 0 = 0$$

$$P2 \ 16 - 2 = 14$$

$$P3 \ 8 - 3 = 5$$

$$\text{Average Wait Time: } (3+0+14+5) / 4 = 5.50$$

### c) Priority Based Scheduling

1. Priority scheduling is a non-pre-emptive algorithm and one of the most common scheduling algorithms in batch systems.
2. Each process is assigned a priority. Process with highest priority is to be executed first and so on.
3. Processes with same priority are executed on first come first served basis.
4. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Wait time of each process is as follows –

Process Wait Time : Service Time - Arrival Time

$$P0 \ 9 - 0 = 9$$

$$P1 \ 6 - 1 = 5$$

$$P2 \ 14 - 2 = 12$$

$$P3 \ 0 - 0 = 0$$

$$\text{Average Wait Time: } (9+5+12+0) / 4 = 6.5$$

### d) Round Robin Scheduling

1. Round Robin is the pre-emptive process scheduling algorithm.
2. Each process is provided a fix time to execute, it is called a quantum.
3. Once a process is executed for a given time period, it is pre-empted and other process executes for a given time period.
4. Context switching is used to save states of pre-empted processes.

Wait time of each process is as follows –

Process Wait Time : Service Time - Arrival Time

$$P0 \ (0 - 0) + (12 - 3) = 9$$

$$P1 \ (3 - 1) = 2$$

$$P2 \ (6 - 2) + (14 - 9) + (20 - 17) = 12$$

$$P3 \ (9 - 3) + (17 - 12) = 11$$

$$\text{Average Wait Time: } (9+2+12+11) / 4 = 8.5$$

### Important Equations:

Turn Around Time = Completion Time – Arrival Time

Waiting Time = Turn Around Time – Burst Time

### PROGRAM:

```
// C program for implementation of FCFS scheduling
#include<stdio.h>
// Function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
    // waiting time for first process is 0
    wt[0] = 0;
```

```

// calculating waiting time
for (int i = 1; i < n ; i++)
    wt[i] = bt[i-1] + wt[i-1] ;
}

// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);

    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    //Display processes along with all details
    printf("Processes  Burst time  Waiting time  Turn around time\n");

    // Calculate total waiting time and total turn around time
    for (int i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf("  %d ",(i+1));
        printf("    %d ", bt[i] );
        printf("    %d",wt[i] );
        printf("    %d\n",tat[i] );
    }
    int s=(float)total_wt / (float)n;
    int t=(float)total_tat / (float)n;
    printf("Average waiting time = %d",s);
    printf("\n");
    printf("Average turn around time = %d ",t);
}

```

```
// Driver code
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    //Burst time of all processes
    int burst_time[] = {10, 5, 8};

    findavgTime(processes, n, burst_time);
    return 0;
}
```

**Output:**

Processes	Burst time	Waiting time	Turn around time
1	10	0	10
2	5	10	15
3	8	15	23

Average waiting time = 8.33333  
Average turn around time = 16

## EXPERIMENT – 2

### FILE ORGANIZATION TECHNIQUES

**AIM:** Simulate the following file organization techniques.

- a. Single level directory.
- b. Two level directory.
- c. Hierarchical.

#### **Single level directory:**

In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory.

#### **Two level directory:**

In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another.

#### **Hierarchical:**

Hierarchical directory structure allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories.

#### **Algorithm for Single Level Directory Structure:**

Step 1: Start

Step 2: Initialize values `gd=DETECT, gm, count, i, j, mid, cir_x;`

Initialize character array `fname[10][20];`

Step 3: Initialize graph function as `Initgraph(& gd, &gm, " c:/tc/bgi");`

`Cleardevice();`

Step 4: set back ground color with `setbkcolor();`

Step 5: read number of files in variable `count.`

Step 6: if check `i < count`

Step 7: for `i=0 & i < count` i increment; `Cleardevice(); setbkcolor(GREEN);` read file name; `setfillstyle(1, MAGENTA);`

Step 8: `mid=640/count; cir_x=mid/3; bar3d(270, 100, 370, 150, 0, 0); settextstyle(2, 0, 4);`  
`settextstyle(1, 1); outtextxy(320, 125, "rootdirectory"); setcolor(BLUE); i++;`

Step 9: for `j=0 & j <= i & cir_x += mid` j increment; `line(320, 150, cir_x, 250);`  
`fillellipse(cir_x, 250, 30, 30); outtextxy(cir_x, 250, fname[i]);`

Step 10: End

**Program Code:**

```
/* Program to simulate single level directory */
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int nf=0,i=0,j=0,ch;
char mdname[10],fname[10][10],name[10];
clrscr();
printf("Enter the directory name:");
scanf("%s",mdname);
printf("Enter the number of files:");
scanf("%d",&nf);
do
{
printf("Enter file name to be created:");
scanf("%s",name);
for(i=0;i<nf;i++)
{
if(!strcmp(name,fname[i]))
break;
}
if(i==nf)
{
strcpy(fname[j++],name);
nf++;
}
else
printf("There is already %s\n",name);
printf("Do you want to enter another file(yes - 1 or no - 0):");
scanf("%d",&ch);
}
while(ch==1);
printf("Directory name is:%s\n",mdname);
printf("Files names are:");
for(i=0;i<j;i++)
printf("\n%s",fname[i]);
getch();
}
```



**Output:**

```
Enter the directory name:sss
Enter the number of files:3
Enter file name to be created:aaa
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:bbb
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:ccc
Do you want to enter another file(yes - 1 or no - 0):0
Directory name is:sss
Files names are:
aaa
bbb
ccc
```

## **EXPERIMENT – 3**

### **BANKERS ALGORITHM**

**AIM:** Implement the banker's algorithm for deadlock avoidance

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

For the Banker's algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request[**MAX**]
2. How much of each resource each process is currently holding[**ALLOCATED**]
3. How much of each resource the system currently has available[**AVAILABLE**]

Resources may be allocated to a process only if the amount of resources requested is less than or equal to the amount available; otherwise, the process waits until resources are available. The Banker's Algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated; otherwise the customer must wait until some other customer deposits enough. Basic data structures to be maintained to implement the

#### **Banker's Algorithm:**

Let  $n$  be the number of processes in the system and  $m$  be the number of resource types. Then we need the following data structures:

1. **Available:** A vector of length  $m$  indicates the number of available resources of each type. If  $\text{Available}[j] = k$ , there are  $k$  instances of resource type  $R_j$  available.
2. **Max:** An  $n \times m$  matrix defines the maximum demand of each process. If  $\text{Max}[i,j] = k$ , then  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .
3. **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process. If  $\text{Allocation}[i,j] = k$ , then process  $P_i$  is currently allocated  $k$  instances of resource type  $R_j$ .
4. **Need:** An  $n \times m$  matrix indicates the remaining resource need of each process. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of resource type  $R_j$  to complete the task.

**PROGRAM:**

```

// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                       { 2, 0, 0 }, // P1
                       { 3, 0, 2 }, // P2
                       { 2, 1, 1 }, // P3
                       { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                     { 3, 2, 2 }, // P1
                     { 9, 0, 2 }, // P2
                     { 2, 2, 2 }, // P3
                     { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
            }
        }
    }
}

```

```
    }  
  }  
  
  if (flag == 0) {  
    ans[ind++] = i;  
    for (y = 0; y < m; y++)  
      avail[y] += alloc[i][y];  
    f[i] = 1;  
  }  
}  
}  
}  
  
printf("Following is the SAFE Sequence\n");  
for (i = 0; i < n - 1; i++)  
  printf(" P%d ->", ans[i]);  
printf(" P%d", ans[n - 1]);  
  
return (0);  
}
```

**Output:**

Following is the SAFE Sequence  
P1 -> P3 -> P4 -> P0 -> P2

## **EXPERIMENT – 4**

### **DISK SCHEDULING ALGORITHM**

**AIM:** Simulate the following disk scheduling algorithms.

a) FCFS b) SCAN c) C-SCAN

#### **ALGORITHM:**

##### **First Come -First Serve (FCFS)**

It is the simplest form of disk scheduling algorithms. The I/O requests are served or processes according to their arrival. The request arrives first will be accessed and served first. Since it follows the order of arrival, it causes the wild swings from the innermost to the outermost tracks of the disk and vice versa. The farther the location of the request being serviced by the read/write head from its current location, the higher the seek time will be.

##### **Example:**

Given the following track requests in the disk queue, compute for the Total Head Movement<sup>2</sup> (THM) of the read/write head: 95, 180, 34, 119, 11, 123, 62, and 64. Consider that the read/write head is positioned at location 50. Prior to this track location 199 was serviced. Show the total head movement for a 200 track disk (0-199).

##### **Solution:**

Total Head Movement Computation:

$$\begin{aligned}(\text{THM}) &= (180-50) + (180-34) + (119-34) + (119-11) + (123-11) + (123-62) + (64-62) \\ &= 130 + 146 + 85 + 108 + 112 + 61 + 2 \quad (\text{THM}) = 644 \text{ tracks}\end{aligned}$$

Assuming a seek rate of 5 milliseconds is given, we compute for the seek time using the formula: Seek Time = THM \* Seek rate = 644 \* 5 ms

$$\text{Seek Time} = 3,220\text{ms}$$

There are some requests that are far from the current location of the R/W head which causes the access arm to travel from innermost to the outermost tracks of the disk or vice versa. In this example, it had a total of 644 tracks and a seek time of 3,220 milliseconds. Based on the result, this algorithm produced higher seek rate since it follows the arrival of the track requests.

##### **SCAN Scheduling Algorithm**

This algorithm is performed by moving the R/W head back-and-forth to the innermost and outermost track. As it scans the tracks from end to end, it process all the requests found in the direction it is headed. This will ensure that all track requests, whether in the outermost, middle or innermost location, will be traversed by the access arm thereby finding all the requests. This is also known as the Elevator algorithm. Using the same sets of example in FCFS the solution are as follows:

$$(\text{THM}) = (50-0) + (180-0) = 50 + 180$$

$$(\text{THM}) = 230$$

$$\text{Seek Time} = \text{THM} * \text{Seek rate} = 230 * 5\text{ms}$$

$$\text{Seek Time} = 1,150\text{ms}$$

This algorithm works like an elevator does. In the algorithm example, it scans down towards the nearest end and when it reached the bottom it scans up servicing the requests that it did not get going down. If a request comes in after it has been scanned, it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks and a seek time of 1,150.

### **Circular SCAN (C-SCAN) Algorithm**

This algorithm is a modified version of the SCAN algorithm. C-SCAN sweeps the disk from end-to-end, but as soon it reaches one of the end tracks it then moves to the other end track without servicing any requesting location. As soon as it reaches the other end track it then starts servicing and grants requests headed to its direction. This algorithm improves the unfair situation of the end tracks against the middle tracks. Using the same sets of example in FCFS the solution are as follows: alpha symbol ( $\alpha$ ) was used to represent the dash line. This return sweeps is sometimes given a numerical value which is included in the computation of the THM. As analogy, this can be compared with the carriage return lever of a typewriter. Once it is pulled to the right most direction, it resets the typing point to the leftmost margin of the report. A typist is not supposed to type during the movement of the carriage return lever because the line spacing is being adjusted. The frequent use of this lever consumes time, same with the time consumed when the R/W head is reset to its starting position.

Assume that in this example,  $\alpha$  has a value of 20ms, the computation would be as follows:

$$(\text{THM}) = (50-0) + (199-62) + \alpha = 50 + 137 + 20$$

$$(\text{THM}) = 207 \text{ tracks}$$

$$\text{Seek Time} = \text{THM} * \text{Seek rate} = 187 * 5\text{ms}$$

$$\text{Seek Time} = 935\text{ms}$$

The computation of the seek time excluded the alpha value because it is not an actual seek or search of a disk request but a reset of the access arm to the starting position.

### **PROGRAM:**

```
// C++ program to demonstrate
// C-SCAN Disk Scheduling algorithm
#include <bits/stdc++.h>
using namespace std;

int size = 8;
int disk_size = 200;

void CSCAN(int arr[], int head)
{
    int seek_count = 0;
    int distance, cur_track;
```

```
vector<int> left, right;
vector<int> seek_sequence;

// appending end values
// which has to be visited
// before reversing the direction
left.push_back(0);
right.push_back(disk_size - 1);

// tracks on the left of the
// head will be serviced when
// once the head comes back
// to the beginning (left end).
for (int i = 0; i < size; i++) {
    if (arr[i] < head)
        left.push_back(arr[i]);
    if (arr[i] > head)
        right.push_back(arr[i]);
}

// sorting left and right vectors
std::sort(left.begin(), left.end());
std::sort(right.begin(), right.end());

// first service the requests
// on the right side of the
// head.
for (int i = 0; i < right.size(); i++) {
    cur_track = right[i];
    // appending current track to seek sequence
    seek_sequence.push_back(cur_track);

    // calculate absolute distance
    distance = abs(cur_track - head);

    // increase the total count
    seek_count += distance;

    // accessed track is now new head
    head = cur_track;
}

// once reached the right end
// jump to the beginning.
```

```
head = 0;

// adding seek count for head returning from 199 to 0
seek_count += (disk_size - 1);

// Now service the requests again
// which are left.
for (int i = 0; i < left.size(); i++) {
    cur_track = left[i];

    // appending current track to seek sequence
    seek_sequence.push_back(cur_track);

    // calculate absolute distance
    distance = abs(cur_track - head);

    // increase the total count
    seek_count += distance;

    // accessed track is now the new head
    head = cur_track;
}

cout << "Total number of seek operations = "
    << seek_count << endl;

cout << "Seek Sequence is" << endl;

for (int i = 0; i < seek_sequence.size(); i++) {
    cout << seek_sequence[i] << endl;
}

// Driver code
int main()
{

    // request array
    int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };
    int head = 50;

    cout << "Initial position of head: " << head << endl;
    CSCAN(arr, head);
```



```
    return 0;  
}
```

**Output**

Initial position of head: 50

Total number of seek operations = 389

Seek Sequence is

60

79

92

114

176

199

0

11

34

41

## EXPERIMENT – 5

### PRODUCER CONSUMER PROBLEM

**AIM:** Implement the producer-consumer problem using semaphores.

#### ALGORITHM

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

1. The producer's job is to generate data, put it into the buffer, and start again.
2. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

**Problem:** To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

**Solution:** One solution of this problem is to use semaphores. The semaphores which will be used here are:

- 1.m, a binary semaphore which is used to acquire and release the lock.
- 2.empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- 3.full, a counting semaphore whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and

The Producer Operation

```
do
{
// wait until empty > 0 and then decrement 'empty'
wait(empty);
// acquire lock
wait(mutex);
/* perform the insert operation in a slot */
// release lock
signal(mutex);
// increment 'full'
signal(full);
}
while(TRUE)
```

1. A producer first waits until there is at least one empty slot.
2. Then it decrements the empty semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
3. Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
4. After performing the insert operation, the lock is released and the value of full is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

```
do
{
// wait until full > 0 and then decrement 'full'
wait(full);
// acquire the lock
wait(mutex);
/* perform the remove operation in a slot */
// release the lock
signal(mutex);
// increment 'empty'
signal(empty);
}
while(TRUE);
```

1. The consumer waits until there is at least one full slot in the buffer.
2. Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
3. After that, the consumer acquires lock on the buffer.
4. Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
5. Then, the consumer releases the lock.
6. Finally, the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

## PROGRAM

```
// C program for the above approach
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
// Initialize a mutex to 1
int mutex = 1;
```

```
// Number of full slots as 0
int full = 0;
```

```
// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;

    // Increase the number of full
    // slots by 1
    ++full;

    // Decrease the number of empty
    // slots by 1
    --empty;

    // Item produced
    x++;
    printf("\nProducer produces "
           "item %d",
           x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
```

```
        "item %d",
        x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}

// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");

    // Using '#pragma omp parallel for'
    // can give wrong value due to
    // synchronisation issues.

    // 'critical' specifies that code is
    // executed by only one thread at a
    // time i.e., only one thread enters
    // the critical section at a given time
    #pragma omp critical

    for (i = 1; i > 0; i++) {

        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
```

```
// is full
else {
    printf("Buffer is full!");
}
break;

case 2:

    // If mutex is 1 and full
    // is non-zero, then it is
    // possible to consume
    if ((mutex == 1)
        && (full != 0)) {
        consumer();
    }

    // Otherwise, print Buffer
    // is empty
    else {
        printf("Buffer is empty!");
    }
    break;

// Exit Condition
case 3:
    exit(0);
    break;
}
}
}
```

## Output:



```
1.Producer
2.Consumer
3.Exit
Enter your choice:2
Buffer is empty!!
Enter your choice:1
Producer produces item 1
Enter your choice:1
Producer produces item 2
Enter your choice:1
Producer produces item 3
Enter your choice:2
Consumer consumes item 3
Enter your choice:1
Producer produces item 3
Enter your choice:2
Consumer consumes item 3
Enter your choice:2
Consumer consumes item 2
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3
```

## EXPERIMENT – 6 DINING PHILOSOPHER'S PROBLEM

**AIM:** Write a program to simulate the working of the dining philosopher's problem.

In computer science, the dining philosopher's problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed. The problem is how to design a discipline of behaviour such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think. The main problem in this case is to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem.

### ALGORITHM

1. Define the number of philosophers
2. Declare one thread per philosopher
3. Declare one semaphore (represent chopsticks) per philosopher
4. When a philosopher is hungry
5. See if chopsticks on both sides are free
6. Acquire both chopsticks or eat
7. restore the chopsticks
8. If chopsticks aren't free
9. Wait till they are available

```
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (ph_num+4)%N
#define RIGHT (ph_num+1)%N
sem_t mutex;
sem_t S[N];
```

```

void * philosopher(void *num);
void take_fork(int);
void put_fork(int);
void test(int);
int state[N];
int phil_num[N]={0,1,2,3,4};
int main()
{
int i;
pthread_t thread_id[N];
sem_init(&mutex,0,1);
for(i=0;i<N;i++)
sem_init(&S[i],0,0);
for(i=0;i<N;i++)
{
pthread_create(&thread_id[i],NULL,philosopher,&phil_num[i]);
printf("Philosopher %d is thinkingn \n",i+1);
}
for(i=0;i<N;i++)
pthread_join(thread_id[i],NULL);
}
void *philosopher(void *num)
{
while(1)
{
int *i = num;
sleep(1);
take_fork(*i);
sleep(0);
put_fork(*i);
}}
void take_fork(int ph_num)
{
sem_wait(&mutex);
state[ph_num] = HUNGRY;
printf("Philosopher %d is Hungryn \n",ph_num+1);
test(ph_num);
sem_post(&mutex);
sem_wait(&S[ph_num]);
sleep(1);}
void test(int ph_num)
{
if (state[ph_num] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
state[ph_num] = EATING;
sleep(2);
}
}

```



```

printf("Philosopher %d takes fork %d and %d \n",ph_num+1,LEFT+1,ph_num+1);
printf("Philosopher %d is Eatingn \n",ph_num+1);
sem_post(&S[ph_num]);
}}
void put_fork(int ph_num)
{
sem_wait(&mutex);
state[ph_num] = THINKING;
printf("Philosopher %d putting fork %d and %d down \n",ph_num+1,LEFT+1,ph_num+1);
printf("Philosopher %d is thinkingn \n",ph_num+1);
test(LEFT);
test(RIGHT);
sem_post(&mutex);
}

```

**OUTPUT**

```

Philosopher 4 is thinkingn
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eatingn
Philosopher 2 is Hungryn
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinkingn
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eatingn
Philosopher 4 is Hungryn
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinkingn
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eatingn
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinkingn
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eatingn
Philosopher 1 is Hungryn
Philosopher 3 is Hungryn
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinkingn
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eatingn
Philosopher 5 is Hungryn
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinkingn
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eatingn
Philosopher 2 is Hungryn
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinkingn
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eatingn
Philosopher 4 is Hungryn
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinkingn
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eatingn
Philosopher 1 is Hungryn
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinkingn

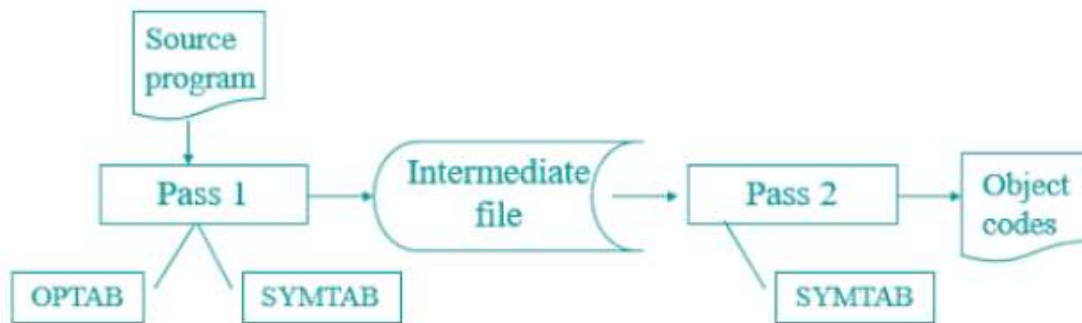
```

## EXPERIMENT – 7

### PASS ONE OF A TWO PASS ASSEMBLER

**AIM:** Implement pass one of a two pass assembler.

Two-pass assembler: Assemblers typically make two or more passes through a source program in order to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction.



Pass 1: Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, name of data fields and programs labels and their relative location (offset) within the segment. . To assign address to labels, the assembler maintains a Location Counter (LC).

Pass 1 determines the amount of code to be generated for each instruction.

#### PROGRAM

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char opcode[10],operand[10],label[10],code[10][10],ch; char
    mnemonic[10][10]={"START","LDA","STA","LDCH","STCH","END"};
    int locctr,start,len,i=0,j=0;
    FILE *fp1,*fp2,*fp3;
    clrscr();
    fp1=fopen("INPUT.DAT","r");
    fp2=fopen("SYMTAB.DAT","w");
    fp3=fopen("OUT.DAT","w");
    fscanf(fp1,"%s%s%s",label,opcode,operand);
    if(strcmp(opcode,"START")==0)
    {
        start=atoi(operand);
    }
}
  
```

```

locctr=start;
fprintf(fp3,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp1,"%s%s%s",label,opcode,operand);
}
else
locctr=0;
while(strcmp(opcode,"END")!=0)
{
fprintf(fp3,"%d",locctr);
if(strcmp(label,"**")!=0)
fprintf(fp2,"%s\t%d\n",label,locctr);
strcpy(code[i],mnemonic[j]);
while(strcmp(mnemonic[j],"END")!=0)
{
if(strcmp(opcode,mnemonic[j])==0)
{
locctr+=3;
break;
}
strcpy(code[i],mnemonic[j]);
j++;
}
if(strcmp(opcode,"WORD")==0)
locctr+=3;
else if(strcmp(opcode,"RESW")==0)
locctr+=(3*(atoi(operand)));
else if(strcmp(opcode,"RESB")==0)
locctr+=(atoi(operand));
else if(strcmp(opcode,"BYTE")==0)
++locctr;
fprintf(fp3,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp1,"%s%s%s",label,opcode,operand);
}
fprintf(fp3,"%d\t%s\t%s\t%s\n",locctr,label,opcode,operand);
fcloseall();
printf("\n\nThe contents of Input Table :\n\n");
fp1=fopen("INPUT.DAT","r");
ch=fgetc(fp1);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp1);
}
printf("\n\nThe contents of Output Table :\n\n\t");
fp3=fopen("OUT.DAT","r");

```

```

ch=fgetc(fp3);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp3);
}
len=locctr-start;
printf("\nThe length of the program is %d.\n\n",len);
printf("\n\nThe contents of Symbol Table :\n\n");
fp2=fopen("SYMTAB.DAT","r");
ch=fgetc(fp2);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp2);
}
fcloseall();
getch();
}

```

**INPUT FILE:****INPUT.DAT**

```

** START 2000
** LDA FIVE
** STA ALPHA
** LDCH CHARZ
** STCH C1
ALPHA RESW 1
FIVE WORD 5
CHARZ BYTE C'Z'
C1 RESB 1
** END **

```

**OUTPUT:**

```

Enter the address where the program has to be loaded : 2000
The contents of output file:
CS_NAME EXT_SYM_NAME ADDRESS LENGTH
PROGA LISTA 2040 0
      ENDA 2054 0
PROGB LISTB 20c3 0
      ENDB 20d3 0
PROGC LISTC 2112 0
      ENDC 2124 0

```

## EXPERIMENT – 8

### PASS TWO OF TWO PASS ASSEMBLER

**AIM:** Implement pass two of a two pass assembler.

Two-pass assembler: Assemblers typically make two or more passes through a source program in order to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction. Pass 2: In the second pass the instructions are again read and are assembled using the symbol table. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are. It produces .OBJ (Object file), .LST (list file) and cross reference (.CRF) files.

### PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char a[10],ad[10],label[10],opcode[10],operand[10],symbol[10],ch; int
    st,diff,i,address,add,len,actual_len,finaddr,prevaddr,j=0;
    char mnemonic[15][15]={"LDA","STA","LDCH","STCH"};
    char code[15][15]={"33","44","53","57"};
    FILE *fp1,*fp2,*fp3,*fp4;
    clrscr();
    fp1=fopen("ASSMLIST.DAT","w");
    fp2=fopen("SYMTAB.DAT","r");
    fp3=fopen("INTERMED.DAT","r");
    fp4=fopen("OBJCODE.DAT","w");
    fscanf(fp3,"%s%s%s",label,opcode,operand);

    while(strcmp(opcode,"END")!=0)
    {
        prevaddr=address;
        fscanf(fp3,"%d%s%s",&address,label,opcode,operand);
```

```

}
finaddr=address;
fclose(fp3);
fp3=fopen("INTERMED.DAT","r");

fscanf(fp3,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{
fprintf(fp1,"\t%s\t%s\t%s\n",label,opcode,operand);
fprintf(fp4,"H^%s^00%s^00%d\n",label,operand,finaddr);
fscanf(fp3,"%d%s%s",&address,label,opcode,operand);
st=address;
diff=prevaddr-st;
fprintf(fp4,"T^00%d^%d",address,diff);
}
while(strcmp(opcode,"END")!=0)
{
if(strcmp(opcode,"BYTE")==0)
{
fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);
len=strlen(operand);
actual_len=len-3;
fprintf(fp4,"^");
for(i=2;i<(actual_len+2);i++)
{
itoa(operand[i],ad,16);
fprintf(fp1,"%s",ad);
fprintf(fp4,"%s",ad);
}
fprintf(fp1,"\n");
}
else if(strcmp(opcode,"WORD")==0)
{
len=strlen(operand);
itoa(atoi(operand),a,10);
fprintf(fp1,"%d\t%s\t%s\t%s\t00000%s\n",address,label,opcode,operand,a);
fprintf(fp4,"^00000%s",a);
}
else if((strcmp(opcode,"RESB")==0)||strcmp(opcode,"RESW")==0))

```

```

fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
else
{
while(strcmp(opcode, mnemonic[j])!=0)
j++;
if(strcmp(operand, "COPY")==0)
fprintf(fp1, "%d\t%s\t%s\t%s\t%s0000\n", address, label, opcode, operand, code[j]);
else
{
rewind(fp2);
fscanf(fp2, "%s%d", symbol, &add);
while(strcmp(operand, symbol)!=0)
fscanf(fp2, "%s%d", symbol, &add);
fprintf(fp1, "%d\t%s\t%s\t%s\t%s%d\n", address, label, opcode, operand, code[j], add);
fprintf(fp4, "^%s%d", code[j], add);
}
}
fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
}
fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
fprintf(fp4, "\nE^00%d", st);
printf("\n Intermediate file is converted into object code");
fcloseall();

printf("\n\nThe contents of Intermediate file:\n\n\t");
fp3=fopen("INTERMED.DAT", "r");
ch=fgetc(fp3);
while(ch!=EOF)
{
printf("%c", ch);
ch=fgetc(fp3);
}
printf("\n\nThe contents of Symbol Table :\n\n");
fp2=fopen("SYMTAB.DAT", "r");
ch=fgetc(fp2);
while(ch!=EOF)
{
printf("%c", ch);
ch=fgetc(fp2);
}

```

```
}
printf("\n\nThe contents of Output file :\n\n");
fp1=fopen("ASSMLIST.DAT","r");
ch=fgetc(fp1);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp1);
}
printf("\n\nThe contents of Object code file :\n\n");
fp4=fopen("OBJCODE.DAT","r");
ch=fgetc(fp4);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp4);
}
fcloseall();
getch();
}
```

### **INPUT FILES:**

#### **INTERMED.DAT**

```
COPY START 2000
2000 ** LDA FIVE
2003 ** STA ALPHA
2006 ** LDCH CHARZ
2009 ** STCH C1
2012 ALPHA RESW 1
2015 FIVE WORD 5
2018 CHARZ BYTE 'EOF'
2019 C1 RESB 1
2020 ** END **
```

#### **SYMTAB.DAT**

```
ALPHA 2012
FIVE 2015
CHARZ 2018
C1 2019
```



**OUTPUT:**

```

Intermediate file is converted into object code
The contents of Intermediate file:

      COPY      START      2000
2000   **      LDA      FIVE
2003   **      STA      ALPHA
2006   **      LDCH     CHARZ
2009   **      STCH     C1
2012   ALPHA    RESW     1
2015   FIVE     WORD     5
2018   CHARZ    BYTE     C' EOF'
2019   C1       RESB     1
2020   **      EMD      **

The contents of Symbol Table :

ALPHA    2012
FIVE     2015
CHARZ    2018
C1       2019

The contents of Output file :

      COPY      START      2000
2000   **      LDA      FIVE      332015
2003   **      STA      ALPHA     442012
2006   **      LDCH     CHARZ     532018
2009   **      STCH     C1        572019
2012   ALPHA    RESW     1
2015   FIVE     WORD     5          000005
2018   CHARZ    BYTE     C' EOF'   454f46
2019   C1       RESB     1
2020   **      EMD      **

The contents of Object code file :

H^COPY^002000^002020
I^002000^19^332015^442012^532018^572019^000005^454f46
E^002000

```

## EXPERIMENT – 9 SINGLE PASS ASSEMBLER

**AIM:** Implement a single pass assembler.

Single Pass Assembler: A single pass assembler scans the program only once and creates the equivalent binary program. The assembler substitute all of the symbolic instruction with machine code in one pass. The difference between one pass and two pass assemblers is basically in the name. A one pass assembler passes over the source file exactly once, in the same pass collecting the labels, resolving future references and doing the actual assembly. The difficult part is to resolve future label references and assemble code in one pass. . A two pass assembler does two passes over the source file (the second pass can be over a file generated in the first pass ). In the first pass all it does is looks for label definitions and introduces them in the symbol table. In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations and so on.

### PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char opcode[10],operand[10],label[10],a[10],ad[10],symbol[10],ch;
    char code[10][10],code1[10][10]={"33","44","53","57"};
    char mnemonic[10][10]={"START","LDA","STA","LDCH","STCH","END"};
    char mnemonic1[10][10]={"LDA","STA","LDCH","STCH"};
    int locctr,start,length,i=0,j=0,k,l=0;
    int st,diff,address,add,len,actual_len,finaddr,prevaddr;
    FILE *fp1,*fp2,*fp3,*fp4,*fp5,*fp6,*fp7;
    clrscr();
    fp1=fopen("INPUT.DAT","r");
    fp2=fopen("SYMTAB.DAT","w");
    fp3=fopen("INETERMED.DAT","w");
    fscanf(fp1,"%s%s%s",label,opcode,operand);
    if(strcmp(opcode,"START")==0)
    {
        start=atoi(operand);
        locctr=start;
        fprintf(fp3,"%s\t%s\t%s\n",label,opcode,operand);
        fscanf(fp1,"%s%s%s",label,opcode,operand);
```

```

}
else
locctr=0;
while(strcmp(opcode,"END")!=0)
{
fprintf(fp3,"%d",locctr);
if(strcmp(label,"**")!=0)
fprintf(fp2,"%s\t%d\n",label,locctr);
strcpy(code[i],mnemonic[j]);
while(strcmp(mnemonic[j],"END")!=0)
{
if(strcmp(opcode,mnemonic[j])==0)
{
locctr+=3;
break;
}
strcpy(code[i],mnemonic[j]);
j++;
}
if(strcmp(opcode,"WORD")==0)
locctr+=3;
else if(strcmp(opcode,"RESW")==0)
locctr+=(3*(atoi(operand)));
else if(strcmp(opcode,"RESB")==0)
locctr+=(atoi(operand));
else if(strcmp(opcode,"BYTE")==0)
++locctr;
fprintf(fp3,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp1,"%s%s%s",label,opcode,operand);
}
fprintf(fp3,"%d\t%s\t%s\t%s\n",locctr,label,opcode,operand);
length=locctr-start;
fcloseall();
printf("\n\nThe contents of Input file:\n\n");
fp1=fopen("INPUT.DAT","r");
ch=fgetc(fp1);
while(ch!=EOF)
{
printf("%c",ch);

```

```

    ch=fgetc(fp1);
}
printf("\n\nLength of the input program is %d.",length);
printf("\n\nThe contents of Symbol Table:\n\n");
fp2=fopen("SYMTAB.DAT","r");
ch=fgetc(fp2);
while(ch!=EOF)
{
    printf("%c",ch);
    ch=fgetc(fp2);
}
fcloseall();
fp4=fopen("ASSMLIST.DAT","w");
fp5=fopen("SYMTAB.DAT","r");
fp6=fopen("INTERMED.DAT","r");
fp7=fopen("OBJCODE.DAT","w");
fscanf(fp6,"%s%s%s",label,opcode,operand);
while(strcmp(opcode,"END")!=0)
{
    prevaddr=address;
    fscanf(fp6,"%d%s%s%s",&address,label,opcode,operand);
}
finaddr=address;
fclose(fp6);
fp6=fopen("INTERMED.DAT","r");
fscanf(fp6,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{
    fprintf(fp4,"\t%s\t%s\t%s\n",label,opcode,operand);
    fprintf(fp7,"H^%s^00%s^00%d\n",label,operand,finaddr);
    fscanf(fp6,"%d%s%s%s",&address,label,opcode,operand);
    st=address;
    diff=prevaddr-st;
    fprintf(fp7,"T^00%d^%d",address,diff);
}
while(strcmp(opcode,"END")!=0)
{
    if(strcmp(opcode,"BYTE")==0)
    {

```

```

fprintf(fp4, "%d\t%s\t%s\t%s\t", address, label, opcode, operand);
len=strlen(operand);
actual_len=len-3;
fprintf(fp7, "^");
for(k=2;k<(actual_len+2);k++)
{
    itoa(operand[k],ad,16);
    fprintf(fp4,"%s",ad);
    fprintf(fp7,"%s",ad);
}
fprintf(fp4, "\n");
}
else if(strcmp(opcode,"WORD")==0)
{
    len=strlen(operand);
    itoa(atoi(operand),a,10);
    fprintf(fp4, "%d\t%s\t%s\t%s\t00000%s\n", address, label, opcode, operand, a);
    fprintf(fp7, "^00000%s", a);
}
else if((strcmp(opcode,"RESB")==0)||strcmp(opcode,"RESW")==0)
    fprintf(fp4, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
else
{
    while(strcmp(opcode,mnemonic1[l])!=0)
        l++;
    if(strcmp(operand,"COPY")==0)
        fprintf(fp4, "%d\t%s\t%s\t%s\t%s0000\n", address, label, opcode, operand, code1[l]);
    else
    {
        rewind(fp5);
        fscanf(fp5, "%s%d", symbol, &add);
        while(strcmp(operand, symbol)!=0)
            fscanf(fp5, "%s%d", symbol, &add);
        fprintf(fp4, "%d\t%s\t%s\t%s\t%s%d\n", address, label, opcode, operand, code1[l], add);
        fprintf(fp7, "^%s%d", code1[l], add);
    }
}
fscanf(fp6, "%d%s%s", &address, label, opcode, operand);
}

```

```
fprintf(fp4, "%d\t%s\t%s\t%s\n",address,label,opcode,operand);
fprintf(fp7, "\nE^00%d",st);
printf("\nObject Program has been generated.");
fcloseall();
printf("\n\nObject Program:\n\n");
fp7=fopen("OBJCODE.DAT","r");
ch=fgetc(fp7);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp7);
}
fcloseall();
getch();
}
```

**INPUT FILE:**

**INPUT.DAT**

COPY START 2000

\*\* LDA FIVE

\*\* STA ALPHA

\*\* LDCH CHARZ

\*\* STCH C1

ALPHA RESW 1

FIVE WORD 5

CHARZ BYTE C'EOF'

C1 RESB 1

\*\* END \*\*

**OUTPUT:**

```
The contents of Input file:
COPY START 2000
** LDA FIVE
** STA ALPHA
** LDCH CHARZ
** STCH C1
ALPHA RESW 1
FIVE WORD 5
CHARZ BYTE C'EOF'
C1 RESB 1
** END **

Length of the input program is 20.

The contents of Symbol Table:
ALPHA    2012
FIVE     2015
CHARZ    2018
C1       2019

Object Program has been generated.

Object Program:
H^COPY^002000^002020
I^002000^19^332015^442012^532018^572019^000005^454f46
E^002000
```

## **EXPERIMENT – 10**

### **TWO PASS MACRO PROCESSOR**

**AIM:** Implement a two pass macro processor

Macro Processor is a program that lets you define the code that is reused many times giving it a specific Macro name and reuse the code by just writing the Macro name only. Generally it doesn't come as a separate program but as a bundle to either assembler or compiler There are three main steps of using a macro

1. Define the macro name
2. Give it's definition
3. Use the macro name from within the program anywhere to use its definition (this step is called macro call)

**Features of macro processor:**

1. Recognize the macro definition.
2. Save macro definition.
3. Recognize the macro call.
4. Perform macro expansion.

Forward reference Problem

The assembler specifies that the macro definition should occur anywhere in the program. So there can be chances of macro call before its definition which gives rise to the forwards reference problem of macro. Due to which macro is divided into two passes:

#### **1. PASS 1-**

Recognize macro definition save macro definition

#### **1. PASS 2-**

Recognize macro call perform macro expansion

**Two-pass macro processor**

Pass1: process all macro definitions

Pass2: expand all macro invocation statements

**Problem**

1. Does not allow nested macro definitions
2. Nested macro definitions

The body of a macro contains definitions of other macros Databases required for pass 2

In pass2 we perform recognize macro call and perform macro expansion

#### **1. COPY FILE**

It is a file it contains the output given from PASS1

#### **2. MNT**

It is used for recognizing macro name

#### **3. MDT**

It is used to perform macro EXPANSION

#### **4. MDTP**

It is used to point to the index of MDT .

The starting index is given by MNT

#### **5. ALA**

It is used to replace the index notation by its actual value



## 6. ESC

**PROGRAM**

```

#include<studio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
FILE *f1,*f2,*f3,*f4,*f5;
void main()
{
char lbl[20],opc[20],opr[20],mname[20],arg[20],check[20];char
ch,dlbl[20],dopc[20],dopr[20];
int c;
clrscr();
f1=fopen("MACIN.DAT","r");
rewind(f1);
f2=fopen("NAMETAB.DAT","r");
rewind(f2);
f3=fopen("DEFTAB.DAT","r");
f4=fopen("EXPAND.DAT","w");
f5=fopen("ARGTAB.DAT","w");
while(!feof(f1))
{
l1:
fscanf(f1,"%s %s %s",lbl,opc,opr);
if(strcmp(opc,mname)==0)
c=1;
if(strcmp(opc,"MACRO")==0)
{
while(strcmp(opc,"MEND")!=0)
{
fscanf(f1,"%s%s%s",lbl,opc,opr);
continue;
}
goto l1;
}
rewind(f2);
rewind(f3);

```

```

fscanf(f2,"%s",mname);
if(strcmp(opc,mname)==0)
{
fprintf(f5," %s",opr);
rewind(f5);
while(!feof(f3))
{
fscanf(f3,"%s%s%s",dlbl,dopc,dopr);
if(strcmp(dopc,"MEND")!=0)
{
if(strcmp(dopc,"MACRO")==0)
{
continue;
}
if(strcmp(dopr,"=X'?1'")==0)
strcpy(dopr,"=X'F1'");
if(strcmp(dopr,"?2,X")==0)
strcpy(dopr,"BUFFER,X");
if(strcmp(dopr,"?3")==0)
strcpy(dopr,"LENGTH");
if(c==1)
{
fprintf(f4," %s\t%s\t%s\n",lbl,opc,opr);
c=0;
}
fprintf(f4," %s\t%s\t%s\n",dlbl,dopc,dopr);
}
}
goto l1;
}
fprintf(f4," %s\t%s\t%s\n",lbl,opc,opr);
}
fcloseall();
printf("\n INPUT\n\n Macro Program before expanded \n");
printf(" ----- \n");
f1=fopen("MACIN.DAT","r");
ch=fgetc(f1);
while(ch!=EOF)
{

```

```

printf("%c",ch);
ch=fgetc(f1);
}
printf("\n Definition Table \n");
printf(" -----\n");
f2=fopen("DEFTAB.DAT","r");
ch=fgetc(f2);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f2);
}
printf("\n Name Table \n");
printf(" -----\n");
f3=fopen("NAMETAB.DAT","r");
ch=fgetc(f3);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f3);
}
getch();
clrscr();
printf("\n\n OUTPUT\n\n Macro Program after expanded \n");
printf(" -----\n\n");
f4=fopen("EXPAND.DAT","r");
ch=fgetc(f4);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f4);
}
printf("\n Argument Table \n");
printf(" -----\n\n");
f5=fopen("ARGTAB.DAT","r");
ch=fgetc(f5);
while(ch!=EOF)
{
printf("%c",ch);

```

```
ch=fgetc(f5);  
}  
fcloseall();  
getch();  
}
```

**INPUT FILE:**

**MACIN.DAT**

```
COPY START NULL  
RDBUFF MACRO INDEV,BUFADR,RECLTH  
NULL CLEAR X  
NULL CLEAR A  
NULL CLEAR S  
NULL +LDT #4096  
NULL TD =X'&INDEV'  
NULL JEQ *-3  
NULL RD =X'&INDEV'  
NULL COMPR A,S  
NULL JEQ *+11  
NULL STCH BUFADR,X  
NULL TIXR T  
NULL JLT *-19  
NULL STX RECLTH  
NULL MEND NULL  
FIRST STL RETADR  
CLOOP RDBUFF F1,BUFFER,LENGTH  
NULL LDA LENGTH  
NULL COMP #0  
NULL JEQ ENDFIL  
EOF BYTE C'EOF'  
THREE WORD 3  
RETADR RESW 1  
LENGTH RESW 1  
BUFFER RESB 4096  
NULL END FIRST
```

**DEFTAB.DAT**

```
COPY START NULL  
RDBUFF MACRO &INDEV,&BUFADR,&RECLTH  
NULL CLEAR X  
NULL CLEAR A  
NULL CLEAR S  
NULL +LDT #4096  
NULL TD =X'?1'
```

```

NULL JEQ *-3
NULL RD =X'?1'
NULL COMPR A,S
NULL JEQ *+11
NULL STCH ?2,X
NULL TIXR T
NULL JLT *-19
NULL STX ?3
NULL MEND NULL
    
```

**NAMETAB.DAT**

RDBUFF

**OUTPUT:**

```

COPY      START      NULL
MDSHIPP   MACRO      INDIR, RDBUFF, RECLTH
NULL      CLEAR      X
NULL      CLEAR      A
NULL      CLEAR      S
NULL      +LDT       @0000
NULL      TD         -X'&INDEX'
NULL      JEQ        -3
NULL      RD         -X'&INDEX'
NULL      COMPR     A,S
NULL      JEQ        *+11
NULL      STCH      BUFFER,X
NULL      TIXR      T
NULL      JLT       *-19
NULL      STX      RECLTH
NULL      MEND      NULL
FIRST     STL        RETADR
CLOOP     RDBUFF    PI, BUFFER, LENGTH
NULL      LDA        LENGTH
NULL      COMP      RR
NULL      JEQ        ENDPI1
EOP       BYTE      C'EOP'
THREE    WORD      3
RETADR    RESW      1
LENGTH   RESW      1
BUFFER    RESB     @0000
NULL     END        FIRST
-----
Definition Table
-----
COPY      START      NULL
MDSHIPP   MACRO      &INDEX, &RDBUFF, &RECLTH
NULL      CLEAR      X
NULL      CLEAR      A
NULL      CLEAR      S
NULL      +LDT       @0000
NULL      TD         -X'?1'
NULL      JEQ        -3
NULL      RD         -X'?1'
NULL      COMPR     A,S
NULL      JEQ        *+11
NULL      STCH      ?2,X
NULL      TIXR      T
NULL      JLT       *-19
NULL      STX      ?3
NULL      MEND      NULL
    
```

```

OUTPUT
Macro Program after expanded
-----
COPY      START      NULL
FIRST     STL        RETADR
CLOOP     RDBUFF    PI, BUFFER, LENGTH
COPY      START      NULL
NULL      CLEAR      X
NULL      CLEAR      A
NULL      CLEAR      S
NULL      +LDT       @0000
NULL      TD         -X'?1'
NULL      JEQ        -3
NULL      RD         -X'?1'
NULL      COMPR     A,S
NULL      JEQ        *+11
NULL      STCH      BUFFER,X
NULL      TIXR      T
NULL      JLT       *-19
NULL      STX      LENGTH
NULL      LDA        LENGTH
NULL      COMP      RR
NULL      JEQ        ENDPI1
EOP       BYTE      C'EOP'
THREE    WORD      3
RETADR    RESW      1
LENGTH   RESW      1
BUFFER    RESB     @0000
NULL     END        FIRST
-----
Argument Table
-----
PI, BUFFER, LENGTH
    
```

## **EXPERIMENT – 11**

### **SINGLE PASS MACRO PROCESSOR**

**AIM:** Implement a single pass macro processor.

A macro instruction is a notational convenience for the programmer. It allows the programmer to write shorthand version of a program (module programming). The macro processor replaces each macro invocation with the corresponding sequence of statements (expanding) One-pass macro processor

1. Every macro must be defined before it is called
2. One-pass processor can alternate between macro definition and macro expansion
3. Nested macro definitions are allowed

The important data structures required in a one-pass macro processor are:

1. DEFTAB (Definition Table): It is a definition table that used to store the macro definition including macro prototype and macro body. Comment lines are not included here, and references to the parameters use a positional notation for efficiency in substituting arguments.
2. NAMTAB (Name Table): This table used for storing macros names. It serves as an index to DEFTAB and maintains pointers that point to the beginning and end of the macro definition in DEFTAB.
3. ARGTAB (Argument Table): It maintains arguments according to their positions in the argument list. During expansion, the arguments from this table substituted for the corresponding parameters in the macro body.

### **PROGRAM**

```
#include<studio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
FILE *f1,*f2,*f3,*f4,*f5;
void main()
{
char lbl[20],opc[20],opr[20],mname[20],arg[20],check[20];char
ch,dlbl[20],dopc[20],dopr[20];
int c;
clrscr();
f1=fopen("MACIN.DAT","r");
rewind(f1);
```

```

f2=fopen("NAMETAB.DAT","r");
rewind(f2);
f3=fopen("DEFTAB.DAT","r");
f4=fopen("EXPAND.DAT","w");
f5=fopen("ARGTAB.DAT","w");
while(!feof(f1))
{
l1:
fscanf(f1,"%s %s %s",lbl,opc,opr);
if(strcmp(opc,mname)==0)
c=1;
if(strcmp(opc,"MACRO")==0)
{
while(strcmp(opc,"MEND")!=0)
{
fscanf(f1,"%s%s%s",lbl,opc,opr);
continue;
}
goto l1;
}
rewind(f2);
rewind(f3);
fscanf(f2,"%s",mname);
if(strcmp(opc,mname)==0)
{
fprintf(f5," %s",opr);
rewind(f5);
while(!feof(f3))
{
fscanf(f3,"%s%s%s",dlbl,dopc,dopr);
if(strcmp(dopc,"MEND")!=0)
{
if(strcmp(dopc,"MACRO")==0)
{
continue;
}
if(strcmp(dopr,"=X'?1'")==0)
strcpy(dopr,"=X'F1'");
if(strcmp(dopr,"?2,X")==0)

```

```

strcpy(dopr,"BUFFER,X");
if(strcmp(dopr,"?3")==0)
strcpy(dopr,"LENGTH");
if(c==1)
{
fprintf(f4," %s\t%s\t%s\n",lbl,opc,opr);
c=0;
}
fprintf(f4," %s\t%s\t%s\n",dlbl,dopc,dopr);
}
}
goto l1;
}
fprintf(f4," %s\t%s\t%s\n",lbl,opc,opr);
}
fcloseall();
printf("\n INPUT\n\n Macro Program before expanded \n");
printf(" ----- \n");
f1=fopen("MACIN.DAT","r");
ch=fgetc(f1);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f1);
}
printf("\n Definition Table \n");
printf(" ----- \n");
f2=fopen("DEFTAB.DAT","r");
ch=fgetc(f2);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f2);
}
printf("\n Name Table \n");
printf(" ----- \n");
f3=fopen("NAMETAB.DAT","r");
ch=fgetc(f3);
while(ch!=EOF)

```



```

{
printf("%c",ch);
ch=fgetc(f3);
}
getch();
clrscr();
printf("\n\n OUTPUT\n\n Macro Program after expanded \n");
printf(" ----- \n\n");
f4=fopen("EXPAND.DAT", "r");
ch=fgetc(f4);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f4);
}
printf("\n Argument Table \n");
printf(" ----- \n\n");
f5=fopen("ARGTAB.DAT", "r");
ch=fgetc(f5);
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(f5);
}
fcloseall();
getch();
}

```

**INPUT FILE:**

**MACIN.DAT**

```

COPY START NULL
RDBUFF MACRO INDEV,BUFADR,RECLTH
NULL CLEAR X
NULL CLEAR A
NULL CLEAR S
NULL +LDT #4096
NULL TD =X'&INDEV'
NULL JEQ *-3
NULL RD =X'&INDEV'
NULL COMPR A,S

```

```
NULL JEQ *+11
NULL STCH BUFADR,X
NULL TIXR T
NULL JLT *-19
NULL STX RECLTH
NULL MEND NULL
FIRST STL RETADR
CLOOP RDBUFF F1,BUFFER,LENGTH
NULL LDA LENGTH
NULL COMP #0
NULL JEQ ENDFIL
EOF BYTE C'EOF'
THREE WORD 3
RETADR RESW 1
LENGTH RESW 1
BUFFER RESB 4096
NULL END FIRST
```

#### **DEFTAB.DAT**

```
COPY START NULL
RDBUFF MACRO &INDEV,&BUFADR,&RECLTH
NULL CLEAR X
NULL CLEAR A
NULL CLEAR S
NULL +LDT #4096
NULL TD =X'?1'
NULL JEQ *-3
NULL RD =X'?1'
NULL COMPR A,S
NULL JEQ *+11
NULL STCH ?2,X
NULL TIXR T
NULL JLT *-19
NULL STX ?3
NULL MEND NULL
```

#### **NAMETAB.DAT**

```
RDBUFF
```

#### **OUTPUT:**

```

Macro Program before expanded
COPY      START      NULL
EDBUFF   MACRO      1MDEV, BUADR, RECLTH
NULL     CLEAR      X
NULL     CLEAR      A
NULL     CLEAR      S
NULL     +LDI       $4096
NULL     TB         -X'&IMDEV'
NULL     JEQ        +-3
NULL     RD         -X'&IMDEV'
NULL     COMPR      A,S
NULL     JEQ        ++11
NULL     STCH       BUADR,X
NULL     TIXR      T
NULL     JLT        +-19
NULL     STX        RECLTH
NULL     MEND
FIRST    SIL        RETADR
CLOOP    RDBUFF     P1, BUFFER, LENGTH
NULL     LDR        LENGTH
NULL     COMP       BB
NULL     JEQ        ENDPIL
EOF      BYTE       C'EOF'
THREE    WORD       3
READR    RESW       1
LENGTH  RESW       1
BUFFER   RESB       4096
NULL     END        FIRST

```

Definition Table

```

COPY      START      NULL
EDBUFF   MACRO      &IMDEV, &BUADR, &RECLTH
NULL     CLEAR      X
NULL     CLEAR      A
NULL     CLEAR      S
NULL     +LDI       $4096
NULL     TB         -X'71'
NULL     JEQ        +-3
NULL     RD         -X'71'
NULL     COMPR      A,S
NULL     JEQ        ++11
NULL     STCH       72,X
NULL     TIXR      T
NULL     JLT        +-19
NULL     STX        73
NULL     MEND

```

```

OUTPUT
Macro Program after expanded
COPY      START      NULL
FIRST    SIL        RETADR
CLOOP    RDBUFF     P1, BUFFER, LENGTH
COPY     START      NULL
NULL     CLEAR      X
NULL     CLEAR      A
NULL     CLEAR      S
NULL     +LDI       $4096
NULL     TB         -X'P1'
NULL     JEQ        +-3
NULL     RD         -X'P1'
NULL     COMPR      A,S
NULL     JEQ        ++11
NULL     STCH       BUFFER,X
NULL     TIXR      T
NULL     JLT        +-19
NULL     STX        LENGTH
NULL     LDR        LENGTH
NULL     COMP       BB
NULL     JEQ        ENDPIL
EOF      BYTE       C'EOF'
THREE    WORD       3
READR    RESW       1
LENGTH  RESW       1
BUFFER   RESB       4096
NULL     END        FIRST

```

Argument Table

```

P1, BUFFER, LENGTH

```

## **EXPERIMENT – 12**

### **ABSOLUTE LOADER**

**AIM:** Implement an absolute loader

#### **ALGORITHM:**

An absolute loader loads a binary program in memory for execution. The binary program is stored in a file contains the following:

1. A Header record showing the load origin, length and load time execution start address of the program.
2. A sequence of binary image records containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the load address of the first byte of this code and a count of the number of bytes of code.
3. The absolute loader notes the load origin and the length of the program mentioned in the header record.
4. It then enters a loop that reads a binary image record and moves the code contained in it to the memory area starting at the address mentioned in the binary image record.
5. At the end, it transfers control to the execution start address of the program.

Advantages of the absolute loading scheme: Absolute Loaders

Simple to implement and efficient in execution. Moreover, saves the memory (core) because the size of the loader is smaller than that of the assembler.

Allows use of multi-source programs written in different languages. In such cases, the given language assembler converts the source program into the language. And a common object file is then prepared by address resolution.

#### **Disadvantages of the absolute loading scheme: Absolute Loaders**

1. The programmer must know and clearly specify to the translator (the assembler) the address in the memory for inner-linking and loading of the programs. Care should take so that the addresses do not overlap.
2. For programs with multiple subroutines, the programmer must remember the absolute address of each subroutine and use it explicitly in other subroutines to perform linking.
3. If the subroutine is modified, the program has to assemble again from first to last.

#### **PROGRAM**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char input[10],label[10],ch1,ch2;
int addr, w=0, start, paddr, l, length=0, end, count=0, k, taddr, address, i=0;
FILE *fp1,*fp2;
void check();
void main()
```

```

{ clrscr();
fp1=fopen("INPUT.dat","r");
fp2=fopen("OUTPUT.dat","w");
fscanf(fp1,"%s",input);
printf("\n\n\t\t\tABSOLUTE LOADER\n");
fprintf(fp2,"\n-----\n");
fprintf(fp2,"MEMORY ADDRESS\t\tCONTENTS");
fprintf(fp2,"\n-----\n");
while(strcmp(input,"E")!=0)
{
if(strcmp(input,"H")==0)
{
fscanf(fp1,"%s %x %x %s",label,&start,&end,input);
address=start;
}
else if(strcmp(input,"T")==0)
{
l=length;
ptaddr=addr;
fscanf(fp1,"%x %x %s",&taddr,&length,input);
addr=taddr;
if(w==0)
{
ptaddr=address;
w=1;
}
for(k=0;k<(taddr-(ptaddr+l));k++)
{
address=address+1;
fprintf(fp2,"xx");
count++;
if(count==4)
{
fprintf(fp2," ");
i++;
if(i==4)
{
fprintf(fp2,"\n\n%x\t\t",address);
i=0;
}
}
}
}

```

```

    }
    count=0;
    }
}
if(taddr==start)
    fprintf(fp2, "\n\n%x\t\t", taddr);
fprintf(fp2, "%c%c", input[0], input[1]);
check();
fprintf(fp2, "%c%c", input[2], input[3]);
check();
fprintf(fp2, "%c%c", input[4], input[5]);
check();
fscanf(fp1, "%s", input);
}
else
{
    fprintf(fp2, "%c%c", input[0], input[1]);
    check();
    fprintf(fp2, "%c%c", input[2], input[3]);
    check();
    fprintf(fp2, "%c%c", input[4], input[5]);
    check();
    fscanf(fp1, "%s", input);
}
}
fprintf(fp2, "\n-----\n");
fcloseall();
printf("\n\n The contents of output file:\n\n");
fp2=fopen("OUTPUT.DAT", "r");
ch2=fgetc(fp2);
while(ch2!=EOF)
{
    printf("%c", ch2);
    ch2=fgetc(fp2);
}
fcloseall();
getch();
}
void check()

```

```
{
count++;
address++;
taddr=taddr+1;
if(count==4)
{
fprintf(fp2," ");
i++;
if(i==4)
{
fprintf(fp2,"\n\n%x\t\t",taddr);
i=0;
}
count=0;
}
}
```

**INPUT FILE:**

**INPUT.DAT**

H COPY 001000 00107A

T 001000 1E 141033 482039 001036 281030 301015 482061 3C1003 00102A 0C1039  
00102D

T 00101E 15 0C1036 482061 081033 4C0000 454F46 000003 000000

T 001047 1E 041030 001030 E0205D 30203F D8205D 281030 302057 549039 2C205E  
38203F

T 001077 1C 101036 4C0000 000000 001000 041030 E02079 302064 509039 DC2079  
2C1036

E 001000

**OUTPUT:**

```

                                ABSOLUTE LOADER

The contents of output file:

-----
MEMORY ADDRESS                CONTENTS
-----
1000          14183348  28398818  36281838  38181548
1010          28613C18  8388182A  8C183988  182D8C18
1020          36482861  8818334C  8888454F  46888883
1030          888888xx  xxxxxxxx  xxxxxxxx  xxxxxxxx
1040          xxxxxxxx  xxxxxx04  18388818  3888285D
1050          38283FD8  285D2818  38382857  5498392C
1060          285E3828  3Fxxxxxx  xxxxxxxx  xxxxxxxx
1070          xxxxxxxx  xxxxxx18  18364C88  88888888
1080          88188884  1838E828  79382864  589839DC
1090          28792C18  36
-----

```



## **EXPERIMENT – 13 RELOCATING LOADER**

**AIM:** Implement a relocating loader

A relocating loader load a program in a designated area of memory, relocates it so that it can execute correctly in that area of memory and passes control to it for execution.

1. The binary program is stored in a file contains the following:

A Header record showing the load origin, length and load time execution start address of the program.

2. Similarly, a sequence of binary image records containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the load address of the first byte of this code and a count of the number of bytes of code.

3. Moreover, a table analogous to RELOCTAB table giving linked addresses of address sensitive instructions in the program.

### **PROGRAM**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void convert(char h[12]);
char bitmask[12];
char bit[12]={0};
void main()
{ char add[6],length[10],input[10],binary[12],relocbit,ch,pn[5];
int start,inp,len,i,address,opcode,addr,actualadd,tlen;
FILE *fp1,*fp2;
clrscr();
printf("\n\n Enter the actual starting address : ");
scanf("%x",&start);
fp1=fopen("RLIN.DAT","r");
fp2=fopen("RLOUT.DAT","w");
fscanf(fp1,"%s",input);
fprintf(fp2," -----\n");
fprintf(fp2," ADDRESS\tCONTENT\n");
fprintf(fp2," -----\n");
```

```

while(strcmp(input,"E")!=0)
{
if(strcmp(input,"H")==0)
{
fscanf(fp1,"%s",pn);
fscanf(fp1,"%x",add);
fscanf(fp1,"%x",length);
fscanf(fp1,"%s",input);
}
if(strcmp(input,"T")==0)
{
fscanf(fp1,"%x",&address);
fscanf(fp1,"%x",&tlen);
fscanf(fp1,"%s",bitmask);
address+=start;
convert(bitmask);
len=strlen(bit);
if(len>=11)
len=10;
for(i=0;i<len;i++)
{
fscanf(fp1,"%x",&opcode);
fscanf(fp1,"%x",&addr);
relocbit=bit[i];
if(relocbit=='0')
actualadd=addr;
else
actualadd=addr+start;
fprintf(fp2, "\n %x\t\t%x%x\n",address,opcode,actualadd);
address+=3;
}
fscanf(fp1,"%s",input);
}
}
fprintf(fp2, " -----\n");
fcloseall();
printf("\n\n The contents of output file (RLOUT.DAT):\n\n");
fp2=fopen("RLOUT.DAT","r");
ch=fgetc(fp2);

```

```
while(ch!=EOF)
{
printf("%c",ch);
ch=fgetc(fp2);
}
fclose(fp2);
getch();
}
void convert(char h[12])
{
int i,l;
strcpy(bit,"");
l=strlen(h);
for(i=0;i<l;i++)
{
switch(h[i])
{
case '0':
    strcat(bit,"0");
    break;
case '1':
    strcat(bit,"1");
    break;
case '2':
    strcat(bit,"10");
    break;
case '3':
    strcat(bit,"11");
    break;
case '4':
    strcat(bit,"100");
    break;
case '5':
    strcat(bit,"101");
    break;
case '6':
    strcat(bit,"110");
    break;
case '7':
```

```
    strcat(bit,"111");
    break;
case '8':
    strcat(bit,"1000");
    break;
case '9':
    strcat(bit,"1001");
    break;
case 'A':
    strcat(bit,"1010");
    break;
case 'B':
    strcat(bit,"1011");
    break;
case 'C':
    strcat(bit,"1100");
    break;
case 'D':
    strcat(bit,"1101");
    break;
case 'E':
    strcat(bit,"1110");
    break;
case 'F':
    strcat(bit,"1111");
    break;
}
}
}
```

**INPUT FILE:**

**RLIN.DAT**

H COPY 001000 00107A

T 001000 1E FFC 14 0033 48 1039 10 0036 28 0030 30 0015 48 1061 3C 0003 20 002A

1C 0039 30 002D

T 002500 15 E00 1D 0036 48 1061 18 0033 4C 1000 80 1000 60 1003

E 000000

**OUTPUT:**

```
Enter the actual starting address : 4000
```

```
The contents of output file (RLOUT.DAT):
```

ADDRESS	CONTENT
4000	144033
4003	485039
4006	104036
4009	204030
400c	304015
400f	485061
4012	3c4003
4015	20402a
4018	1c4039
401b	30402d
5500	1d4036
5503	485061
5506	104033
5509	4c1000
550c	001000
550f	601003

## **EXPERIMENT – 14**

### **SYMBOL TABLE USING HASHING**

**AIM:** Implement a symbol table with suitable hashing

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

1. To store the names of all entities in a structured form at one place.
2. To verify if a variable has been declared.
3. To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
4. To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table.

#### **Symbol table using hashing**

1. In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables.
2. A hash table is an array with index range: 0 to table size – 1. These entries are pointer pointing to names of symbol table.
3. To search for a name we use hash function that will result in any integer between 0 to table size – 1.
4. Insertion and lookup can be made very fast –  $O(1)$ .
5. Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

#### **ALGORITHM**

1. Start processing.
2. Declare structure for input and output files
3. Declare File pointers for input and output files
4. Open Input File(s) in Read mode and Open Output File(s) in write Mode.
5. Read the Intermediate File until EOF occurs.
  - 5.1 If Symbol is not equal to - then
  - 5.2 Generate the hash index for the symbol using the hash function.
  - 5.3 Write the Symbol Name and its address into Symbol table.
6. Close all the Files.
7. Print Symbol Table is created.
8. Stop processing.

**PROGRAM**

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
>
#include<conio.h>
#define size 50
struct intermediate
{
int addr;
char label[10];
char mnem[10];
char op[10];
}res;
struct symbol
{
char symbol[10];
int addr;
}sy,syy[size];
int hs=size;
unsigned hash(char *);
void search();
void main()
{
FILE
*s1,*p1;
char *lb;
int as=0,i=0;
clrscr();
s1=fopen("source.txt","r");
p1=fopen("symbol.txt","w");
while(!feof(s1))
{
fscanf(s1,"%d %s %s
%s",&res.addr,res.label,res.mnem,res.op);
strcpy(lb,res.label);
if(strcmp(res.label,"
-
")!=0)
{
as=ha
sh(lb);
strcpy(syy[as].symbol,res.label);
syy[as].addr=res.addr;
strcpy(sy.symbol,res.label);

```

```

sy.addr=res.addr;
fprintf(p1,"%s
\
t%d
\
n",sy.symbol,sy.addr);
}
}
fcloseall();
printf("Symbol Table
\
n");
printf("Index
\
tLabel
\
tAddr
\
n");
for(i=0;i<size;i++)
{
if(syy[i].addr!=0)
printf("%d
\
t%s
\
t%d
\
n",i,syy[i].symbol,syy[i].addr);
}
search();
}
unsigned hash(char *s)
{
unsigned hashval;
for (hashval = 0; *s != '
\
0'; s++)
hashval = *s + hashval;
return hashval % hs;
}
void sea
rch()
{
char *name;

```



```

char ch;
int val=0;
do
{
printf("Do u want to search label using hashing:(y/n)");
ch=getche();
if(ch=='y')
{
printf("
\
nEnter the label to be searched:");
scanf("%s",name);
val=hash(name);
if(strcmp(syy[val].symbol,name)!=0)
val=0;
if(val!=0)
printf("Label %s found in hash table at index %d
with value
%d
\
n",syy[val].symbol,val,syy[val].addr);
else
printf("Label not found
\
n");
}
}while(ch!='n');
getch();
}

```

Input & Output:

Input File:

SOURCE.TXT

1000

start

lda

copy

1003

endfil

add

0050

1006

copy

resw

1

```
1009
first
stl
endfil
1012
-
end
start
Output File:
SYMBOL.TXT
start
1000
endfil
1003
copy
1006
first
1009
Symbol Table
Index
Label
Addr
17
copy
1006
24
start
1000
36
first
1009
48
endfil
1003
Do u want to search label using hashing:(y/n)y
Enter the label to be searched:start
Label start found in hash table at index 24 with value 1000
Do u want to search label using hashing:(y/n)y
Enter the label to be searched:cloop
Label not found
Do u want to search label using hashing:(y/n)n
```