MUTHOOT INSTITUTE OF TECHNOLOGY AND SCIENCE
VARIKOLI, ERNAKULAM-682308

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

CS 331 SYSTEM SOFTWARE LAB

(AY:2019-20)

LAB MANUAL
Version number: 1.3

Prepared by:

Dr. Resmi N.G.
Ms. Dhanya Sudarsan
Ms. Asha Raj
Assistant Professor
CSE Department

Changes from the previous version

1. **Added description on each experiment.**
2. **Added additional experiment on page replacement algorithms.**
3. **Added a new set of sample viva questions.**

SYLLABUS

Course code	Course Name	L-T-P Credits	Year of Introduction
CS331	SYSTEM SOFTWARE LAB	0-0-3-1	2016

Prerequisite: Nil

Course Objectives

- To build an understanding on design and implementation of different types of system software.

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.

List of Exercises/Experiments: (Exercises/experiments marked with * are mandatory from each part. Total 12 Exercises/experiments are mandatory)

Part A

1. Simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.
a) FCFS b) SJF c) Round Robin (pre-emptive) d) Priority
2. Simulate the following file allocation strategies.
a) Sequential b) Indexed c) Linked
3. Implement the different paging techniques of memory management.
4. Simulate the following file organization techniques *
a) Single level directory b) Two level directory c) Hierarchical
5. Implement the banker's algorithm for deadlock avoidance.*
6. Simulate the following disk scheduling algorithms. *
a) FCFS b)SCAN c) C-SCAN
7. Simulate the following page replacement algorithms
a) FIFO b)LRU c) LFU
8. Implement the producer-consumer problem using semaphores. *
9. Write a program to simulate the working of the dining philosopher's problem.*

Part B

10. Implement the symbol table functions: create, insert, modify, search, and display.
11. Implement pass one of a two pass assembler. *
12. Implement pass two of a two pass assembler. *
13. Implement a single pass assembler. *
14. Implement a two pass macro processor *
15. Implement a single pass macro processor.
16. Implement an absolute loader.
17. Implement a relocating loader.
18. Implement pass one of a direct-linking loader.
19. Implement pass two of a direct-linking loader.
20. Implement a simple text editor with features like insertion / deletion of a character, word, and sentence.
21. Implement a symbol table with suitable hashing.*

TABLE OF CONTENTS

EXPERIMENT 1

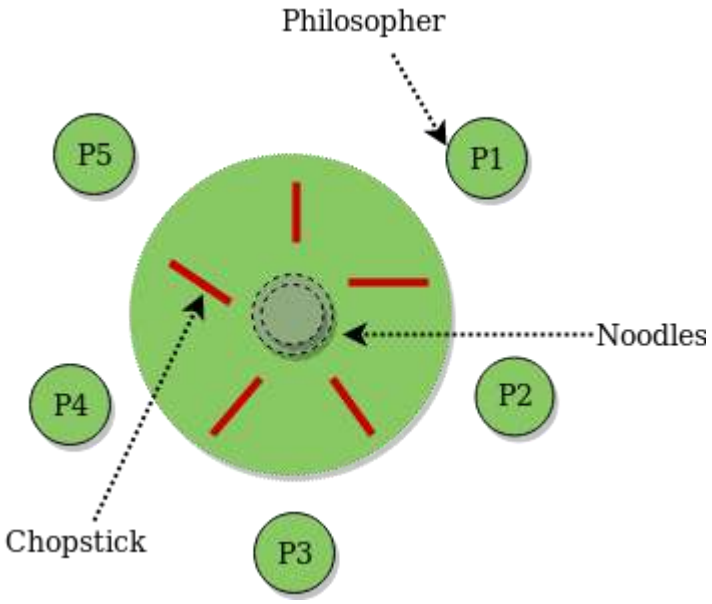
SIMULATE DINING PHILOSOPHER'S PROBLEM

AIM

To simulate Dining Philosopher's problem in C language.

DESCRIPTION

The Dining Philosopher Problem – The dining philosopher's problem is a classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes. It assumes that 5 philosophers (represents processes) are seated around a circular table with one chopstick (represents a resource) between each pair of philosophers. A philosopher spends her life thinking and eating. In the center of the table is a bowl of noodles. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the chopsticks that are between her and her left and right neighbours. A philosopher may pick up only one chopstick at a time. A philosopher may eat if she can pick up the two chopsticks adjacent to her. She eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



Semaphore Solution to Dining Philosopher – There are three states of philosopher : THINKING, HUNGRY and EATING. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

ALGORITHM

1. START
2. Declare the number of philosophers.
3. Declare a semaphore mutex to acquire and release lock on each chopstick.
4. Declare one semaphore per chopstick.

5. Create a thread for each philosopher.
6. Set initial state of each philosopher to THINKING.
7. For each philosopher, repeat the following steps:
    a. Set the state to HUNGRY.
    b. Try to acquire the left and right chopsticks.
    c. If both chopsticks are available, then:
        i. Set the state to EATING.
        ii. Perform eating.
        iii. Release the chopsticks.
        iv. Signal the waiting philosophers, if any.
    d. Otherwise, wait till both the chopsticks are available.
8. STOP

OUTPUT

Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 2 is Hungry
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry

RESULT

The Dining Philosopher's problem is simulated and output is verified.

EXPERIMENT 2

SIMULATE PRODUCER CONSUMER PROBLEM

AIM

To simulate producer consumer problem in C language.

DESCRIPTION

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, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is 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.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores.

A semaphore S is an integer variable that can be accessed only through two standard operations: wait() and signal(). The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1. In this solution, we use one binary semaphore – mutex and two counting semaphores – full and empty. "Mutex" is for acquiring and releasing the lock on shared buffer. "Full" keeps track of number of items in the buffer at any given time and "empty" keeps track of number of unoccupied slots.

Initialization of semaphores
mutex = 1
full = 0 // Initially, all slots are empty. Thus full slots are 0
empty = n // All slots are empty initially

Solution for Producer
When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Solution for Consumer
As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

AGORITHM:

1. Declare 3 semaphores mutex, full, empty.
2. Initialize mutex to 1, full to 0 and empty to n.

3. Create separate thread for each producer and consumer.

Producer

1. Produce the item.
2. Wait on empty (if all n buffers are full) until a consumer consumes item from the buffer and signals the producer.
3. Acquire mutex lock to access the shared buffer.
4. Insert item into buffer.
5. Release the mutex lock.
6. Signal (on full) the waiting consumer when a buffer is full.

Consumer

1. Wait on full (if no buffer is full) until a producer inserts item into the buffer and signals the consumer.
2. Acquire the mutex lock to access the shared buffer.
3. Consume item from the buffer.
4. Release the mutex lock.
5. Signal (on empty) the waiting producer when a buffer is empty.

OUTPUT

Producer produces 1 item
Consumer consumes 1 item
Producer produces 1 item
Producer produces 2 item
Producer produces 3 item
Producer produces 4 item
Consumer consumes 4 item
Producer produces 4 item
Producer produces 5 item
Producer produces 6 item
Producer produces 7 item
Producer produces 8 item
Consumer consumes 8 item
Producer produces 8 item
Producer produces 9 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item
Producer produces 10 item
Buffer is full

Consumer consumes 10 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item
Producer produces 10 item
Buffer is full
Consumer consumes 10 item

RESULT

The Producer consumer's problem is simulated and output is verified.

EXPERIMENT 3

SIMULATE CPU SCHEDULING ALGORITHMS

AIM

To simulate the following CPU Scheduling techniques:

a) First Come First Serve (FCFS)
b) Shortest Job First (SJF)
c) Round Robin scheduling
d) Priority scheduling

DESCRIPTION

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

1. FCFS

First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

Completion Time: Time at which process completes its execution.

Turn Around Time (Total time): Time difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time

Waiting Time: Time difference between turn around time and burst time.
Waiting Time = Turn Around Time – Burst Time (Service time)

2. SJF

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm. It has the advantage of having minimum average waiting time among all scheduling algorithms. It is a greedy algorithm. It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging. It is practically infeasible as operating system may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

3. Round Robin Scheduling

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is simple, easy to implement, and starvation-free as all processes get fair share of CPU. One of the most commonly used technique in CPU scheduling as a core. It is preemptive as processes are assigned CPU only for a fixed slice of time at most. The disadvantage of it is more overhead of context switching.

4. Priority Scheduling

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

ALGORITHM

FCFS

1. Get the number of processes.
2. Get the ID and service time for each process.
3. Initially, waiting time of first process is zero and total time for the first process is the service time of that process.
4. Calculate the total time and processing time for the remaining processes as follows:
    a. Waiting time of a process is the total time of the previous process.
    b. Total time of a process is calculated by adding its waiting time and service time.
5. Total waiting time is calculated by adding the waiting time of each process.
6. Total turn around time is calculated by adding the total time of each process.
7. Calculate average waiting time by dividing the total waiting time by total number of processes.
8. Calculate average turn around time by dividing the total turn around time by the number of processes.
9. Display the result.

SJF

1. Get the number of processes.
2. Get the ID and service time for each process.
3. Initially the waiting time of first short process is set as 0 and total time of first short process is taken as the service time of that process.
4. Calculate the total time and waiting time of remaining processes as follows:
    a. Waiting time of a process is the total time of the previous process.
    b. Total time of a process is calculated by adding the waiting time and service time of each process.
5. Total waiting time is calculated by adding the waiting time of each process.
6. Total turn around time is calculated by adding the total time of each process.
7. Calculate average waiting time by dividing the total waiting time by total number of processes.
8. Calculate average turn around time by dividing the total turn around time by total number of processes.
9. Display the result.

Round Robin

1. Get the number of processes.
2. Get the process id, burst time and arrival time for each of the processes.
3. Create an array rem_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array).
4. Create another array wt[] to store waiting times of processes. Initialize this array as 0.
5. Initialize time : t = 0
6. Find waiting time for each process by traversing all the processes while all processes are not done. Do the following for i-th process if it is not done yet.
    a. If rem_bt[i] > quantum
        i) t = t + quantum
        ii) bt_rem[i] = bt_rem[i] - quantum;

b. else // Last cycle for this process
   i) t = t + bt_rem[i];
   ii) wt[i] = t - bt[i]
   iii) bt_rem[i] = 0; // This process is over

7. Compute turn around time for each process i as:
   tat[i] = wt[i] + bt[i]
8. Total waiting time is calculated by adding the waiting time of each process.
9. Total turn around time is calculated by adding the total time of each process.
10. Calculate average waiting time by dividing the total waiting time by total number of processes.
11. Calculate average turn around time by dividing the total turn around time by total number of processes.
12. Display the result.

Priority Scheduling

1. Get the number of processes.
2. Get the process id, service time and priority for each process.
3. Sort the processes according to the priority.
4. Now, apply FCFS algorithm:
   a. Initially, waiting time of first process is zero and total time for the first process is the service time of that process.
   b. Calculate the total time and processing time for the remaining processes as follows:
      i. Waiting time of a process is the total time of the previous process.
      ii. Total time of a process is calculated by adding its waiting time and service time.
5. Total waiting time is calculated by adding the waiting time of each process.
6. Total turn around time is calculated by adding the total time of each process.
7. Calculate average waiting time by dividing the total waiting time by total number of processes.
8. Calculate average turn around time by dividing the total turn around time by total number of processes.
9. Display the result.

OUTPUT

FCFS

```
Enter total number of processes(maximum 20):3

Enter Process Burst Time
P[1]:24
P[2]:3
P[3]:3

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            24              0               24
P[2]            3               24              27
P[3]            3               27              30

Average Waiting Time:17
Average Turnaround Time:27
Process returned 0 (0x0)   execution time : 7.661 s
Press any key to continue.
```

SJF

```
Enter number of process:4

Enter Burst Time:
p1:4
p2:8
p3:3
p4:7

Process      Burst Time          Waiting Time     Turnaround Time
p3                3                   0                3
p1                4                   3                7
p4                7                   7                14
p2                8                   14               22

Average Waiting Time=6.000000
Average Turnaround Time=11.500000

Process returned 35 (0x23)   execution time : 5.567 s
Press any key to continue.
```

Round Robin Scheduling

```
Average Waiting Time= 5.250000
Avg Turnaround Time = 9.500000tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$ .
.out
Enter Total Process:     4
Enter Arrival Time and Burst Time for Process Process Number 1 :0
9
Enter Arrival Time and Burst Time for Process Process Number 2 :1
5
Enter Arrival Time and Burst Time for Process Process Number 3 :2
3
Enter Arrival Time and Burst Time for Process Process Number 4 :3
4
Enter Time Quantum:     5


Process |Turnaround time|waiting time

P[2]    |       9       |       4
P[3]    |       11      |       8
P[4]    |       14      |       10
P[1]    |       21      |       12

Average Waiting Time= 8.500000
Avg Turnaround Time = 13.750000tusharsoni@tusharsoni-Lenovo-G50-70:~/Desktop$
```

Priority Scheduling

```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:6
Priority:3

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:14
Priority:1

P[4]
Burst Time:6
Priority:4

Process      Burst Time          Waiting Time     Turnaround Time
P[3]              14                  0                14
P[2]              2                   14               16
P[1]              6                   16               22
P[4]              6                   22               28

Average Waiting Time=13
Average Turnaround Time=20
```

RESULT

The CPU scheduling algorithms are simulated and output is verified.

EXPERIMENT 4

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM

To implement Banker's Algorithm for deadlock avoidance in C language.

DESCRIPTION

The **Banker's 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.

Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$.

Max :

- It is a 2-d array of size '**nxm'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'nxm'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'nxm'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation$_i$ specifies the resources currently allocated to process P_i and **Need**$_i$ specifies the additional resources that process P_i may still request to complete its task.
Banker's algorithm consists of Safety algorithm and Resource request algorithm.

ALGORITHM

Safety Algorithm
The algorithm is for finding out whether or not a system is in a safe state and can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
   Initialize: Work = Available
   Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both
   a) Finish[i] = false
   b) Need$_i$ <= Work

If no such i exists goto step (4).

3) Work = Work + $Allocation_i$
Finish[i] = true
Goto step (2)

4)If Finish [i] = true for all i then
the system is in a safe state.

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2) If $Request_i$ <= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.
3) Assuming that the system has allocated the requested resources to process P_i modifying the state as follows:
Available = Available – $Request_i$
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$ – $Request_i$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$, and the od resource-allocation state is restored.

OUTPUT

Enter the number of resources: 4

Enter the number of processes: 5

Enter the process id for process initiating request: 3

Enter Claim Vector: 8 5 9 7

Enter Allocated Resource Table: 2 0 1 1 0 1 2 1 4 0 0 3 0 2 1 0 1 0 3 0

Enter Maximum Claim table: 3 2 1 4 0 2 5 2 5 1 0 5 1 5 3 0 3 0 3 3

The Claim Vector is: 8 5 9 7

The Allocated Resource Table:

2 0 1 1

0 1 2 1

4 0 0 3

0 2 1 0

1 0 3 0

The Maximum Claim Table:

3 2 1 4

0 2 5 2

```
5    1    0    5
1    5    3    0
3    0    3    3
```

Allocated resources: 7 3 7 5

Available resources: 1 2 2 2

Process3 is executing.

The process is in safe state.

Available vector: 5 2 2 5

Process1 is executing.

The process is in safe state.

Available vector: 7 2 3 6

Process2 is executing.

The process is in safe state.

Available vector: 7 3 5 7

Process4 is executing.

The process is in safe state.

Available vector: 7 5 6 7

Process5 is executing.

The process is in safe state.

Available vector: 8 5 9 7

RESULT

The Banker's algorithm for deadlock avoidance is simulated and output is verified.

EXPERIMENT 5

SIMULATE FILE ORGANIZATION TECHNIQUES

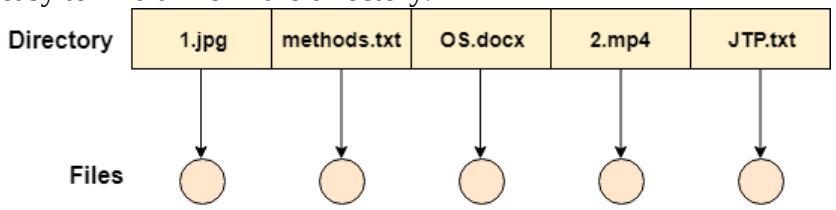
AIM

To implement the following file organization techniques:

a) Single level directory
b) Two level directory
c) Hierarchical

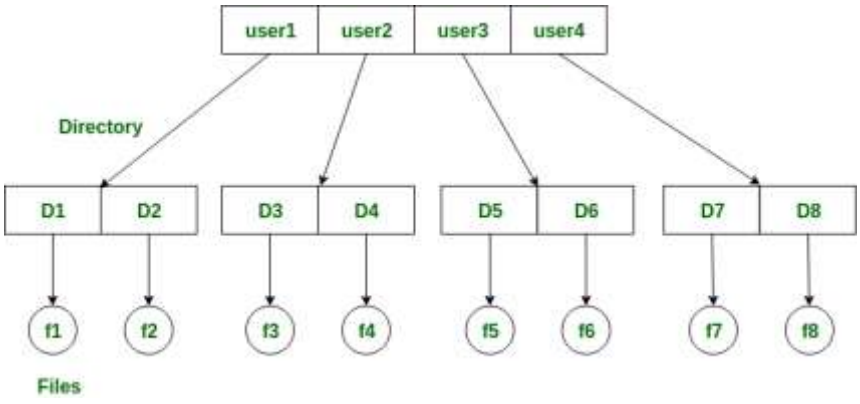
DESCRIPTION

The directory structure is the organization of files into a hierarchy of folders. 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.

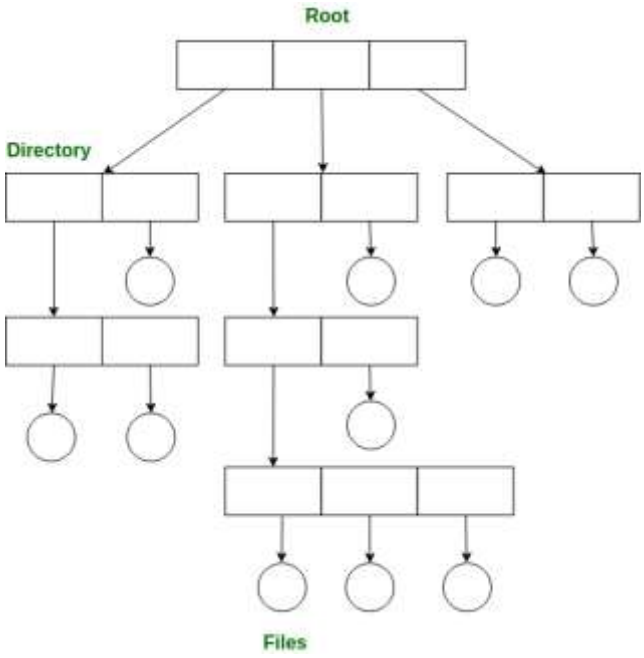


Single Level Directory

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 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

a) Single level directory

1. Create a directory structure to store a single directory and multiple files.
2. Enter a directory name.
3. Perform the following operations:
    a. Create file
        i. Accept filename.
        ii. Increment file count.
        iii. Update the file information table.
    b. Delete file
        i. Accept the name of file to be deleted.
        ii. Compare the filename with names of existing files.
        iii. If a match is found, then
            1. Delete the file by updating file information table.
            2. Decrement the file count.
        iv. Otherwise, display 'File not found!'.
    c. Search file
        i. Accept the name of file to be searched.
        ii. Compare the filename with names of existing files.
        iii. If a match is found, then
            1. Display 'File is found!'
        iv. Otherwise, display 'File not found!'.
    d. Display files
        i. Check if directory is empty.
        ii. If yes, print 'Directory empty!'.
        iii. Otherwise, print information of each file from the file information table.
4. EXIT

b) Two level directory

1. Create a structure to store details of multiple directories and multiple files for each of the directories.
2. Perform the following operations:
    a. Create directory
        i. Accept the directory name.
        ii. Increment the directory count.
        iii. Set file count as 0.
        iv. Update the directory information table.
        v. Display 'Directory created!'.
    b. Create file
        i. Accept the directory name.
        ii. Compare the directory name with names of existing directories.
        iii. If match is found, then
            1. Accept the filename.
            2. Increment file count for this directory.
            3. Update corresponding file information table.
        iv. Otherwise, print 'Directory not found!'.
    c. Delete file
        i. Accept the directory name.
        ii. Compare the directory name with names of existing directories.
        iii. If match is found, then
            1. Accept the name of file to be deleted.
            2. Compare the filename with names of existing files in this directory.
            3. If a match is found, then
                a. Delete the file by updating corresponding file information table.
                b. Decrement the file count for this directory.
            4. Otherwise, display 'File not found!'.
        iv. Otherwise, display 'Directory not found!'
    d. Search file
        i. Accept the directory name.
        ii. Compare the directory name with names of existing directories.
        iii. If match is found, then
            1. Accept the name of file to be deleted.
            2. Compare the filename with names of existing files in this directory.
            3. If a match is found, then
                a. Display 'File found!'
            4. Otherwise, display 'File not found!'.
        iv. Otherwise, display 'Directory not found!'
    e. Display files
        i. Accept the directory name.
        ii. Compare the directory name with names of existing directories.
        iii. If match is found, then
            1. Check if directory is empty.
            2. If yes, then
                a. Display 'Directory empty!'
            3. Otherwise, display the information of files in that directory.
        iv. Otherwise, display 'Directory not found!'.
3. EXIT

c) Hierarchical

1. Create a root directory.
2. Create subdirectories and files under root directory as required.
3. Create subdirectories and files under each subdirectory as required resulting in a tree structure.
4. Display the tree structure.

OUTPUT

Single level:

Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 4

The Files are -- A B C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC

File ABC not found
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B
File B is deleted

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 5

Two Level

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1
Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR2

Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A2
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR2
Enter name of the file -- B1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 5
Directory Files
DIR1 A1 A2
DIR2 B1

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 4

Enter name of the directory – DIR
Directory not found

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 3
Enter name of the directory – DIR1
Enter name of the file -- A2

File A2 is deleted

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice – 6

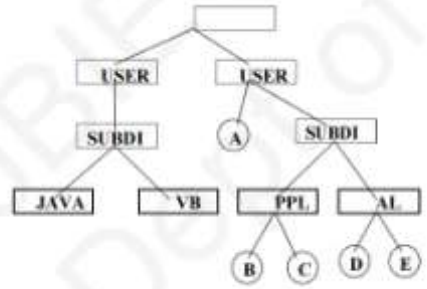
Hierarchical

Enter Name of dir/file (under root): ROOT
Enter 1 for Dir / 2 For File : 1
No of subdirectories / files (for ROOT) :2
Enter Name of dir/file (under ROOT):USER 1
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for USER 1):1
Enter Name of dir/file (under USER 1):SUBDIR
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for SUBDIR):2
Enter Name of dir/file (under USER 1):
JAVA Enter 1 for Dir /2 for file:1
No of subdirectories /files (for JAVA): 0
Enter Name of dir/file (under SUBDIR):VB
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for VB): 0

Enter Name of dir/file (under ROOT):USER2
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for USER2):2
Enter Name of dir/file (under ROOT):A
Enter 1 for Dir /2 for file:2
Enter Name of dir/file (under USER2):SUBDIR 2
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for SUBDIR 2):2

Enter Name of dir/file (under SUBDIR2):PPL
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for PPL):2
Enter Name of dir/file (under PPL):B
Enter 1 for Dir /2 for file:2
Enter Name of dir/file (under PPL):C
Enter 1 for Dir /2 for file:2
Enter Name of dir/file (under SUBDIR):AI
Enter 1 for Dir /2 for file:1
No of subdirectories /files (for AI): 2
Enter Name of dir/file (under AI):D
Enter 1 for Dir /2 for file:2
Enter Name of dir/file (under AI):E

Enter 1 for Dir /2 for file:2



RESULT

Single level, two-level and hierarchical directory structures were simulated and the programs executed successfully.

EXPERIMENT 6

SIMULATE DISK SCHEDULING ALGORITHMS

AIM

To implement the following disk scheduling algorithms:
a) FCFS – First Come First Serve
b) SCAN
c) C-SCAN

DESCRIPTION

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

a) FCFS

FCFS is the simplest of all the disk scheduling algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

- Every request gets a fair chance.
- No indefinite postponement of any request.

Disadvantages:

- Does not try to optimize the seek time.
- May not provide the best possible service.

b) SCAN

In SCAN algorithm, the disk arm moves in a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm

c) C-SCAN

In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area. These situations are avoided in C-SCAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Advantages:

- Provides more uniform wait time compared to SCAN.

ALGORITHM

FCFS:

1. Let request array represent an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. One by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Add this distance to the total head movement.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

SCAN:

1. Let request array represent an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represent whether the head is moving towards left or right.
3. In the direction in which head is moving, service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Add this distance to the total head movement.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until one of the ends of the disk is reached.
8. If end of the disk is reached, reverse the direction and go to step 2 until all tracks in request array have not been serviced.

C-SCAN:

1. Let request array represent an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represent whether the head is moving towards left or right.
3. In the direction in which head is moving, service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Add this distance to the total head movement.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until one of the ends of the disk is reached.
8. If end of the disk is reached, set the head position at the other end and go to step 3 until all tracks in request array have not been serviced.

OUTPUT

```
 Enter the maximum number of cylinders : 200
Enter number of queue elements8
 Enter the work queue95
180
34
119
11
123
62
64

 Enter the disk head starting position:
50
```

```
                MENU
                1. FCFS
                2. SCAN
                3. C-SCAN
                4. EXIT

Enter your choice: 1
FCFS
*****
 Total seek time : 644

                MENU
                1. FCFS
                2. SCAN
                3. C-SCAN
                4. EXIT

Enter your choice: 3
 Enter the previous disk head position:
199

 C-SCAN
*****
 Total seek time : 386
                MENU
                1. FCFS
                2. SCAN
                3. C-SCAN
                4. EXIT

Enter your choice: 3
 Enter the previous disk head position:
5

 C-SCAN
*****
 Total seek time : 382

                MENU
                1. FCFS
                2. SCAN
                3. C-SCAN
                4. EXIT

Enter your choice: 2
 Enter the previous disk head position:
50

 SCAN
*****
 Total seek time : 230
```

MENU
1. FCFS
2. SCAN
3. C-SCAN
4. EXIT

Enter your choice:4

RESULT

Thus the C program to implement the disk scheduling algorithms namely FCFS, SCAN and C-SCAN algorithms was written and executed successfully. The obtained outputs were verified.

PART – B

EXPERIMENT 7

IMPLEMENTATION OF PASS ONE OF A TWO PASS ASSEMBLER

AIM

To write a program to implement pass one of a two pass assembler.

DESCRIPTION

Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader. It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here, assembler divides these tasks in two passes:

Pass-1:

a) Define symbols and literals and remember them in symbol table and literal table respectively.
b) Keep track of location counter
c) Process pseudo-operations

ALGORITHM

```
Begin
        read first input line;
        if OPCODE = 'START' then
            begin
                save #[OPERAND] as starting address
                initialized LOCCTR to starting address
                write line to intermediate file
                read next input line
            end {if START}
        else
            initialize LOCCTR to 0
        while OPCODE != 'END'
            do begin
                if this is not a comment line then
                begin
                    if there is a symbol in the LABEL field then begin
                        search SYMTAB for LABEL
                        if found then
                            set error flag (duplicate symbol)
                        else
                            insert (LABEL, LOCCTR) into SYMTAB
                    end {if symbol}
                    search OPTAB for OPCODE
                    if found then
                        add 3 {instruction length} to LOCCTR
```

```
                else if OPCODE = 'WORD' then
                        add 3 to LOCCTR
                else if OPCODE = 'RESW' then
                        add 3 * #[OPERAND] to LOCCTR
                else if OPCODE = 'RESB' then
                        add #[OPERAND] to LOCCTR
                else if OPCODE = 'BYTE' then
                        begin
                                find length of constant in bytes
                                add length to LOCCTR
                         end {if BYTE}
                else
                        set error flag (invalid operation code)
            end {if not a comment}
            write line to intermediate file
            read next input line
        end {while not END}
        write last line to intermediate file
        save (LOCCTR - starting address) as program length
end
```

OUTPUT

INPUT FILES:

input.txt

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

optab.txt

```
START   *
LDA   03
STA 0f
LDCH 53
STCH 57
END *
```

OUTPUT FILES:

The length of the program is 20

symtab.txt

ALPHA 2012
FIVE 2015
CHARZ 2018
C1 2019

ouput.txt

	**	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'Z'
2019	C1	RESB	1
2020	**	END	**

RESULT

The pass 1 of a two pass assembler was simulated successfully and the output was verified.

EXPERIMENT 8

IMPLEMENTATION OF PASS TWO OF A TWO PASS ASSEMBLER

AIM

To write program to implement pass two of a two pass assembler.

DESCRIPTION

Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration (machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table).

Pass-2:

a) Generate object code by converting symbolic op-code into respective numeric op-code.

b) Generate data for literals and look for values of symbols

ALGORITHM

```
Begin
    read first input file {from intermediate file}
    if OPCODE = 'START' then
    begin
        write listing line
        read next input line


    end {if START}
    write header record to object program
    initialize first text record
    while OPCODE != 'END' do
     begin
         if this is not a comment line then
         begin
             search OPTAB for OPCODE
             if found then
              begin
                  if there is a symbol in OPERAND field then
                  begin
                      search SYMTAB for OPERAND
                      if found then
                          store symbol value as operand address
                      else
                          begin
                              store 0 as operand address
                              set error flag (undefined symbol)
                           end
                  end {if symbol}
```

```
                    else
                        store 0 as operand address
                    assemble the object code instruction
                end {if opcode found}
                else if OPCODE = 'BYTE' or 'WORD' then
                    convert constant to object code
                if object code not fit into the current Text record then
                begin
                    write Text record to object program
                    initialize new Text record
                end
                add object code to Text record
                end {if not comment}
            write listing line
            read next input line
        end {while not END}
        write last Text record to object program
        write End record to object program
        write last listing line
end
```

OUTPUT

INPUT FILES:

out.txt

```
        **      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'Z'
2019    C1      RESB    1
2020    **      END     **
```

optab.txt

```
START   *
LDA     03
STA     0f
LDCH    53
STCH    57
END     *
```

symtbl.txt

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

OUTPUT FILES:

twoout.txt

**	START		2000	
**	LDA	FIVE	2000	032015
**	STA	ALPHA	2003	0f2012
**	LDCH	CHARZ	2006	532018
**	STCH	C1	2009	572019
ALPHA	RESW	1	2012	
FIVE	WORD	5	2015	5
CHARZ	BYTE	C'Z'	2018	5a
C1	RESB	1	2019	
**	END	**	2020	

RESULT

Pass 2 of a two pass assembler was simulated successfully and the output was verified.

EXPERIMENT 9

IMPLEMENTATION OF A SINGLE PASS ASSEMBLER

AIM

To write a program to implement a single pass assembler.

DESCRIPTION

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. One-pass assemblers are used when it is necessary or desirable to avoid a second pass over the source program the external storage for the intermediate file between two passes is slow or is inconvenient to use.
Main problem: forward references to both data and instructions.
One simple way to eliminate this problem: require that all areas be defined before they are referenced. It is possible, although inconvenient, to do so for data items. Forward jump to instruction items cannot be easily eliminated.

ALGORITHM

Step no	Details of the step
1	Begin
2	Read first input line if OPCODE='START' then a. save #[operand] as starting address
3	b. initialize LOCCTR as starting address c. read next input line end
4	else initialize LOCCTR to 0 while OPCODE != 'END' do d. if there is not a comment line then e. if there is a symbol in the LABEL field then i. search SYMTAB for LABEL ii. if found then
5	1. if symbol value as null 2. set symbol value as LOCCTR and search the linked list with the corresponding operand 3. PTR addresses and generate operand addresses as corresponding symbol values 4. set symbol value as LOCCTR in symbol table and delete the linked list iii. end iv. else insert (LABEL,LOCCTR) into SYMTAB v. end
6	search OPTAB for OPCODE
7	if found then search SYMTAB for OPERAND address
8	if found then

f. if symbol value not equal to null then
i) store symbol value as operand address
else insert at the end of the linked list with a node with address as LOCCTR
9 else insert (symbol name, null) add 3 to LOCCTR.
10 elseif OPCODE='WORD' then
add 3 to LOCCTR & convert comment to object code
11 elseif OPCODE = 'RESW' then add 3 #[OPERND] to LOCCTR
12 elseif OPCODE = 'RESB' then
add #[OPERND] to LOCCTR
elseif OPCODE = 'BYTE' then
13 g. find length of the constant in bytes
h. add length to LOCCTR
convert constant to object code
if object code will not fit into current text record then
14 i. write text record to object program
j. initialize new text record
o. add object code to text record
15 write listing line
16 read next input line
17 write last text record to object program
18 write end record to object program
19 write last listing line
20 End

OUTPUT

INPUT FILES:

input1.tx t

```
**      START   6000
**      JSUB    CLOOP
**      JSUB    RLOOP
ALPHA   WORD    23
BETA    RESW    3
GAMMA   BYTE    C'Z'
DELTA   RESB    4
CLOOP   LDA     ALPHA
RLOOP   STA     BETA
**      LDCH    GAMMA
**      STCH    DELTA
**      END     **
```

optab1.txt

```
*
START
JSUB    48
LDA     14
STA     03
LDCH    53
STCH    57
END     *
```

OUTPUT FILES:

CLOOP 6023
RLOOP 6026
ALPHA 6006
BETA 6009
GAMMA 6018
DELTA 6019

spout.txt

**	START	6000	0
**	JSUB	CLOOP	486023
**	JSUB	RLOOP	486026
ALPHA	WORD	23	23
BETA	RESW	3	0
GAMMA	BYTE	C'Z'	90
DELTA	RESB	4	0
CLOOP	LDA	ALPHA	146006
RLOOP	STA	BETA	036009
**	LDCH	GAMMA	536018
**	STCH	DELTA	576019
**	END	**	0

RESULT

A single pass assembler was implemented successfully and the output was verified.

EXPERIMENT 10

IMPLEMENTATION OF A MACRO PROCESSOR

AIM

To write a program to implement a macro processor.

DESCRIPTION

In assembly language programming it is often that some set or block of statements get repeated every now. In this context the programmer uses the concept of macro instructions (often called as macro) where a single line abbreviation is used for a set of line. For every occurrence of that single line the whole block of statements gets expanded in the main source code. This gives a high level feature to assembly language that makes it more convenient for the user to write code easily.

A macro instruction (macro) is simply a notational convenience for the programmer. It allows the programmer to write shorthand version of a program (module programming). A macro represents a commonly used group of statements in the source program. The macro processor replaces each macro instruction with the corresponding group of source statements. This operation is called "expanding the macro" Using macros allows a programmer to write a shorthand version of a program. For example, before calling a subroutine, the contents of all registers may need to be stored. This routine work can be done using a macro.

Features of macro processor:

- Recognized the macro definition
- Save macro definition
- Recognized the macro call
- 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:

PASS 1-

Recognize macro definition save macro definition

PASS 2-

Recognize macro call perform macro expansion

Data Structures used:

(1) DEFTAB - Stores the macro definition including macro prototype and macro body. Comment lines are omitted. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

(2) NAMTAB - Stores macro names, which serves an index to DEFTAB contain pointers to the beginning and end of the definition.

(3) ARGTAB - Used during the expansion of macro invocations. When a macro invocation statement is encountered, the arguments are stored in this table according to their position in the argument list.

ALGORITHM

Step no.	Details of the step
1	Start the macro processor program.
2	Include the necessary header files and variable.
3	Open the three files a.f1=macin.dat with read privilege b. f2=macout.dat with write privilege c. f3= deftab.dat with write privilege
4	Get the variable form f1 file macin.dat for label,opcode,operand
5	Read the variable until the opcode is not is equal to zero
6	Then check if the opcode is equal to Macro if Macro Then Copy macroname=label a. Get the variable label, opcode, operand b. In these if condition perform the while loop until opcode is not equal to MEND c. Copy the variable d. close while loop and if condition e. else if opcode is equal to macro name Perform the for loop from 0 to length
7	Finally terminate the program.

OUTPUT

INPUT FILE:

macin.txt

```
**      macro  m1
**      move a,b
**      mend   ---
**      macro  m2
**      lda b
**      mend   ---
**      start 1000
**      lda a
**      call m1
**      call m2
**      add a,b
```

OUPUT FILE:

```
No. of macros=2
Enter the text filename
outmac
Macin
```

outmac.txt

```
**       macro m1
**       move  a,b
**      mend   ---
**      macro  m2
**      lda              b
**      mend   ---
```

```
**      start   1000
**      lda             a
**      move            a,b
**      lda             b
```

RESULT

A macro processor was implemented successfully and the output was verified.

EXPERIMENT 11

IMPLEMENTATION OF A SYMBOL TABLE

AIM

To write a program to implement symbol table.

DESCRIPTION

A Symbol table is a data structure used by the compiler, where each identifier in program's source code is stored along with information associated with it relating to its declaration. It stores identifier as well as it's associated attributes like scope, type, line-number of occurrence, etc.

ALGORITHM

1. Initialize all the variables.
2. Create a symbol table with fields as variable and value using create option. Entries may be added to the table while it is created itself.
3. Perform operations as insert, modify, search and display.
4. Append new contents to the symbol table with the constraint that there is no duplication of entries, using insert option.
5. Modify existing content of the table using modify option.
6. Use display option to display the contents of the table.

OUTPUT

```
1.CREATE
2.INSERT
3.MODIFY
4.SEARCH
5.DISPLAY
6.EXIT:1

Enter the no. of entries:2
Enter the variable and the values:-
a 23
c 45

The table after creation is:
Variable      value
a             23
c             45
1.CREATE 2.INSERT 3.MODIFY 4.SEARCH 5.DISPLAY 6.EXIT:2

Enter the variable and the value:b 34
The table after insertion is:
Variable      value
a             23
c             45
b             34
1.CREATE 2.INSERT 3.MODIFY 4.SEARCH 5.DISPLAY 6.EXIT: 3
```

Enter the variable to be modified:c

The current value of the variable c is 45. Enter the new variable and its value 44 The table after modification is:

Variable	value
a	23
c	44
b	34

1.CREATE 2.INSERT 3.MODIFY 4.SEARCH 5.DISPLAY 6.EXIT: 4

Enter the variable to be searched:a
The location of the variable is 1
The value of a is 23.

1.CREATE 2.INSERT 3.MODIFY 4.SEARCH 5.DISPLAY 6.EXIT: 5

Variable	value
a	23
c	44
b	34

RESULT

The symbol table was implemented successfully and the output was verified.

EXPERIMENT 12

IMPLEMENTING SYMBOL TABLE WITH HASHING

AIM

To implement a symbol table with suitable hashing in C language.

DESCRIPTION

Symbol table can be implemented using various data structures like:

- LinkedList
- Hash Table
- Tree

A common data structure used to implement a symbol table is HashTable.

ALGORITHM

1. Create m number of nodes in a symbol table, where m is the size of the table.
2. Perform operations, insert, search and display.
3. Add entries to the symbol table by converting the string to be inserted as integers and applying the hash function - key mod m.
4. Entries may be added to the table while it is created itself.
5. Append new contents to the symbol table using hash function with the constraint that there is no duplication of entries, using "insert" option.
6. Modify existing contenst of the table using modify option.
7. Use display option to display the contents of the table.

OUTPUT

Enter your choice:

1.create a symbol table
2.search in the symbol table
1
Enter the address:1000
enter the label:system
Continue(y/n)?y

Enter the address:2000
enter the label: software
Continue(y/n)?n

The Symbol Table is

hash values	address	label
0	0	
1	0	

```
2               0
3               0
4               0
5               0
6               0
7               0
8               0
9           2000 software
10          1000 system

enter your choice:
1.create a symbol table
2.search in the symbol table
2
Enter the label:system
The label — system — is present in the symbol table at address:1000

enter your choice:
 1.create a symbol table
 2.search in the symbol table
2
Enter the label: cse
The label is not present in the symbol table
```

RESULT

The symbol table with suitable hashing was implemented in C language, and the output has been verified.

ADDITIONAL EXPERIMENTS

SIMULATING PAGE REPLACEMENT ALGORITHMS

AIM

To implement the following page replacement algorithms:

a) First In First Out (FIFO)
b) Least Recently Used (LRU)

DESCRIPTION

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms

- **First In First Out (FIFO)**
  This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

- **Least Recently Used (LRU)**
  In this algorithm, the page which is least recently used is replaced.

ALGORITHM
1. FIFO

1. Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the pages in the queue to perform FIFO.

c) Increment page fault.

ii) Else

If current page is present in set, do nothing.

Else

a) Remove the first page from the queue as it was the first to be entered in the memory.

b) Replace the first page in the queue with the current page in the string.

c) Store current page in the queue.

d) Increment page faults.

2. Return the number of page faults.

2. LRU

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1. Start traversing the pages.

i) If set holds less pages than capacity

a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.

b) Simultaneously maintain the recent occurred index of each page in a map called indexes.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Find the page in the set that was least recently used using index array. Replace the page with minimum index with current page.

b) Increment page faults.

c) Update index of current page.

2. Return number of page faults.

OUTPUT

Enter data

Enter length of page reference sequence:20

Enter the page reference sequence:7 2 3 1 2 5 3 4 6 7 7 1 0 5 4 6 2 3 0 1

Enter no of frames:3

Page Replacement Algorithms
1.FIFO
2.LRU
3.Exit
Enter your choice:1

For 7 : 7
For 2 : 7 2
For 3 : 7 2 3
For 1 : 2 3 1
For 2 :No page fault

```
For 5 : 3 1 5
For 3 :No page fault
For 4 : 1 5 4
For 6 : 5 4 6
For 7 : 4 6 7
For 7 :No page fault
For 1 : 6 7 1
For 0 : 7 1 0
For 5 : 1 0 5
For 4 : 0 5 4
For 6 : 5 4 6
For 2 : 4 6 2
For 3 : 6 2 3
For 0 : 2 3 0
For 1 : 3 0 1
Total no of page faults:17
Enter data

Enter length of page reference sequence:10

Enter the page reference sequence:1 2 3 2 4 1 3 2 4 1

Enter no of frames:3

Page Replacement Algorithms
 1.FIFO
2.LRU
3.Exit
Enter your choice:2

For 1 : 1
For 2 : 1 2
For 3 : 1 2 3
For 2 :No page fault!
For 4 : 4 2 3
For 1 : 4 2 1
For 3 : 4 3 1
For 2 : 2 3 1
For 4 : 2 3 4
For 1 : 2 1 4
Total no of page faults:9
```

RESULT

The program to implement FIFO and LRU page replacement algorithms was executed successfully and the output was verified.

SAMPLE VIVA QUESTIONS
PART – A

1. List the Process States.
– New State – means a process is being created
– Running – means instructions are being executed
– Waiting – means a process is waiting for certain conditions or events to occur
– Ready – means a process is waiting for an instruction from the main processor
– Terminate – means a process is stopped abruptly

2. What is a short-term scheduler ?
It selects which process has to be executed next and allocates CPU.

3. What is preemptive and non-preemptive scheduling algorithm?
A scheduling algorithm which allow a process to be temporarily suspended is called preemptive scheduling and the algorithm that do not allow any process to be suspended till its completion is called non-preemptive scheduling.

4. What are different performance metrics for scheduler?
There are five performance metrics for scheduler, those are as follows:
- CPU Utilization: Percentage of time that the CPU is doing useful work.
- Waiting time: Average time a process spends for its turn to get executed.
- Throughput: Number of processes completed / time unit.
- Response Time: Average time elapsed from when process is submitted until useful output is obtained.
- Turnaround Time: Average time elapsed from when process is submitted to when it has completed.

5. What is mutex?
Mutex is a locking mechanism which allows only one process to access the resource at a time. It stands for mutual exclusion and ensures that only one process can enter the critical section at a time.

6. What is critical section of code?
As the name suggests, it is that section or part of code which has variables or other resources which is being shared by two processes, and if allowed to access may lead to race condition. To avoid this, we need a mutex which ensures only one process can enter the critical section of code at a time.

7. How is synchronization achieved by semaphore?
There are two operations which are performed on semaphore which helps in synchronization, those are:
- Wait: If the semaphore value is not negative, decrement the value by 1.
- Signal: Increments the value of semaphore by 1.

8. What is deadlock?
A deadlock is a situation where two or more processes or threads sharing the same resources are effectively preventing each other from accessing the resources. Thus, none of the process can continue executing leading to deadlock.

9. What are the conditions required for deadlock to happen?
There are four conditions required for deadlock which are stated below

Mutual Exclusion: At least one resource is held in a non-sharable mode that is only one process at a time can use the resource. If another process requests that resource, the requesting process has to wait till it is released.

Hold and Wait: There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

No Preemption: Resources cannot be preempted; that is, a resource can only be released after the process has completed its task.

Circular Wait: There must exist a set {p0, p1,…..pn} of waiting processes such that p0 is waiting for a resource which is held by p1, p1 is waiting for a resource which is held by p2,…, pn-1 is waiting for a resource which is held by pn and pn is waiting for a resource which is held by p0.

10. What is race condition?
Race condition is an undesirable situation where two or more threads/process are trying to access and change the value of data shared by them and the outcome of the operation depends upon the order in which the execution happens. Thus it lead to data inconsistency and loss of its value as both threads are racing to change/access the shared data.

PART - B

1. Define System Software.
System software is a type of computer program that is designed to run a computer's hardware and application programs.

2. Define the basic functions of an assembler.
a) Translating mnemonic operation codesto their machine language equivalents.
b) Assigning machine addresses to symbolic labels used by the programmer.

3. What are forward references?
It is reference to a label that is defined later in a program.

4. What are the 3 different records used in an object program?
a) Header record – contains program name, starting address and length of the program.
b) Text record – contains translated instructions and data of the program.
c) End record - marks the end of the object program and specifies the address in the program where execution is to begin.

5. What is the need of SYMTAB in assembler?
The symbol table includes the name and value for each symbol in the source program, together with flags to indicate error conditions.

6. What is the need of OPTAB in assembler?
The operation code table contains the mnemonic code and its machine language equivalent.

7. What are the symbol defining statements generally used in assemblers?
a) EQU – to define symbols and specify their values directly.
b) ORG – to indirectly assign values to symbols.

8. What is the use of the variable LOCCTR in assembler?
This variable is used to assign addresses to symbols. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR and hence whenever we reach a label in the source program, the current value of LOCCTR gives the address associated with the label.

9. Define load and go assembler.
One pass assembler that generates the object code in memory for immediate execution is known as load and go assembler.

10. Differentiate between the assembler directives RESW and RESB.
RESW reserves the indicated number of words for data area. RESB reserves the indicated number of bytes for data area.

SAMPLE LAB EXAM QUESTIONS

1) Write a program to simulate producer-consumer problem with multiple producers and consumers.
2) Implement FCFS CPU scheduling algorithm.
3) Implement C-SCAN disk scheduling algorithm.
4) Write a program to print the contents of DEFTAB, NAMTAB and ARGTAB.
5) Write a program to count all the forward references in a given program.
6) Write a program to print all the forward references in a given program.
7) Implement Pass 1 of a two pass assembler.