SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 OF THE UGC ACT, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

CAP412

OPERATING SYSTEMS LAB

I MCA

SCHOOL OF COMPUTING

Course Objective

To enable students to acquire knowledge on various concepts of operating systems by doing hands on experiments on Linux commands *process management*, *concurrency, synchronization, inter-process communication*, *file management*, *processor management* and *memory management , disk scheduling*

LIST OF EXPERIMENTS

Linux Shell Commands

1. Write a shell script to know the information about a file.
2. Write a shell script to locate lines with only two words and replace

   All the lines with two words with the given input

3. Write a shell script to replace a file with its sorted version.
4. Write a shell script that accepts the name of a text file and finds the

   number of sentences, number of words, and number of words that start with a vowel.

LIST OF EXPERIMENTS

Processor Scheduling

5. Write a program to implement the following process scheduling algorithms
   a) FCFS b) SJF

6. Write a program to implement the following process scheduling algorithms

(a)Priority (b) Round Robin

File allocation and Organization

7. Write a program to implement the following file allocation strategies
   a) Sequential b) Linked c) Indexed Sequential

8. Write a program to Implement File Organization Techniques

a)Single level directory b) Two level directory

Process concurrency

9. Write a program to implement the concept of semaphores.

LIST OF EXPERIMENTS

Memory Management- Paging and page replacement algorithms

10. Implement the page replacement algorithms

a) FIFO b) LRU

11. Write a program to implement Paging Technique of memory management.

Disk Scheduling Algorithms

12. Write a program to implement the following disk scheduling algorithms

a) FIFO b) SCAN c) C-SCAN d) LOOK

1.Information About a File

Objective:

To write a program to simulate the link command like cat, head and tail and ls command

Procedure:

1. Start the process.
2. In the cat file get the lines from the user and redirect to the file.
3. In the shell command in the file read the lines and print the first ten lines
   in the file.
4. In the shell command provide the details like file extension, file name, date creation etc..
5. Stop the Process

2. Replacing a line with two words

Objective:

To locate lines with only two words and replace them with the input given by the user

Procedure:

1. Start the process
2. Read the file name
3. Check whether it is read ,write and executable(i.e) using if statement.
4. Get the number of lines using WC command.
5. Use While statement to start with the first line of the program.
6. Within the loop check for lines with only two words(i.e) using WC –c command and replace them with the input.
7. Stop the Process

3. Replacing a file with its sorted version .

Objective:

To replace the given file with its sorted version.

Procedure:

1. Start the process
2. Read the file to be replaced.
3. Display the contents of the file using cat command.
4. Redirect the output of cat command as input to sort command.
5. Redirect the output to a file.
6. Display the contents of the file(i.e)sorted version of the input file.
7. Stop the Process

4. Manipulation of text file

Objective:

To write a shell script that accepts name of a text file and finds the number of sentences, number of words, number of words that start with vowel.

Procedure:

1. Start
2. Read the file name.
3. If the file is available then go to the process otherwise write a message that enter the valid file name and it should be displayed.
4. Assign path terminal = “ttu”.
5. Redirect the file to terminal exec<$filename.
6. Using while loop split into the one line in the file.
7. Send the variable into the switch case and find articles, vowels, words, sentences more than 5 letters.
8. Redirect the control to the keyboard.
9. Result will be printed.
10. Do you wish continue. Press ‘Y’ else the program will be terminated.
11. Stop

5. Process scheduling
a. FCFS

Objective:

To implement the first come first serve without arrival time CPU scheduling algorithm

Procedure:

Step 1: Create the number of process.
Step 2: Get the ID and Service time for each process.
Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.
Step 4: Calculate the Total time and Processing time for the remaining processes.
Step 5: Waiting time of one process is the Total time of the previous process.
Step 6: Total time of process is calculated by adding Waiting time and Service time.
Step 7: Total waiting time is calculated by adding the waiting time for lack process.
Step 8: Total turn around time is calculated by adding all total time of each process.
Step 9: Calculate Average waiting time by dividing the total waiting time by total number of process.
Step 10: Calculate Average turn around time by dividing the total time by the number of process.
Step 11: Display the result

5 b. Process Scheduling SJF.

Objective:

To implement the CPU scheduling algorithm for shortest job first.

Procedure:

Step 1: Get the number of process.
Step 2: Get the id and service time for each process.
Step 3: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.
Step 4: Calculate the total time and waiting time of remaining process.
Step 5: Waiting time of one process is the total time of the previous process.
Step 6: Total time of process is calculated by adding the waiting time and service time of each process.
Step 7: Total waiting time calculated by adding the waiting time of each process.
Step 8: Total turn around time calculated by adding all total time of each process.
Step 9: Calculate average waiting time by dividing the total waiting time by total number of process.
Step 10: Calculate average turn around time by dividing the total waiting time by total number of process.
Step 11: Display the result.

6 a. CPU Scheduling using Priority .

Objective:

To implement the CPU scheduling algorithm using Priority.

Procedure

Step 1: Get the number of process

Step 2: Get the id and service time for each process.

Step 3: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step 4: Calculate the total time and waiting time of remaining process.

Step 5: Waiting time of one process is the total time of the previous process.

Step 6: Total time of process is calculated by adding the waiting time and service time of each process.

Step 7: Total waiting time calculated by adding the waiting time of each process.

Step 8: Total turn around time calculated by adding all total time of each process.

Step 9: Calculate average waiting time by dividing the total waiting time by total number of process.

Step 10: Calculate average turn around time by dividing the total waiting time by total number of process.

Step 11: Display the result.

6 b. CPU Scheduling Round robin

Objective:

To write the program to simulate the Round Robin program.

Procedure

Step 1: Initialize all the structure elements

Step 2: Receive inputs from the user to fill process id, burst time and arrival time.

Step 3: Calculate the waiting time for all the process id.

- The waiting time for first instance of a process is calculated as: a[i].waittime=count + a[i].arrivt
- The waiting time for the rest of the instances of the process is calculated as:
  - If the time quantum is greater than the remaining burst time then waiting time is calculated as:

    a[i].waittime=count + tq
  - Else if the time quantum is greater than the remaining burst time then waiting time is calculated as:

    a[i].waittime=count - remaining burst time

Step 4: Calculate the average waiting time and average turnaround time

Step 5: Print the results of the step 4.

7a. File allocation using sequential access

Objective:

To write a C program to implement File Allocation concept using sequential method

Procedure:

1. Input size of the process to be allocated memory
2. Allocate memory to the process using the placement algorithm. If memory of that size available then raise an exception.
3. When a request for terminating a process received, release the memory of that process and designate it as free memory.
4. Compute the memory size

7b. File allocation using Linked list.

Objective:

To write a C program to implement File Allocation concept using the technique Linked List Technique..

Procedure:

Step 1: Start the Program

Step 2:Obtain the required data through char and int datatypes.

Step 3:Enter the file name, starting block ending block.

Step 4: Print the free block using loop.

Step 5: for loop is created to print the file utilization of linked type of entered type .

Step 6: This is allocated to the unused linked allocation.

Step 7: Stop the execution.

7c. File allocation using Indexing technique.

Objective:

To write a C program to implement File Allocation concept using the technique indexed allocation Technique..

Procedure:

Step 1: Start the Program

Step 2:Obtain the required data through char and int datatypes.

Step 3:Enter the filename , index block.

Step 4: Print the file name index loop.

Step 5:Fill is allocated to the unused index blocks

Step 6: This is allocated to the unused linked allocation.

Step 7: Stop the execution.

8. Implementation of Producer/Consumer problem usingSemaphores

Objective:

To write a C program to implement the Producer & consumer Problem (Semaphore)

Procedure:

Step 1: The Semaphore mutex, full & empty are initialized.

Step 2: In the case of producer process

i) Produce an item in to temporary variable.
ii) If there is empty space in the buffer check the mutex value for enter into the critical section.
iii) If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

i) It should wait if the buffer is empty
ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
iii) Signal the mutex value and reduce the empty value by 1.
iv) Consume the item.

Step 4: Print the result

9. File Organization Techniques using single level directory and two level directory

Objective:

To write a C program to implement File Organization concept using the technique Single level directory.

Procedure:(Single level)

Step 1: Start the Program
Step 2:Obtain the required data through char and int datatypes.
Step 3:Enter the filename,index block.
Step 4: Print the file name index loop.
Step 5:Fill is allocated to the unused index blocks
Step 6: This is allocated to the unused linked allocation.
Step 7: Stop the execution

Procedure:(Two level)

Step 1: Start the Program
Step 2: Obtain the required data through char and in datatypes.
Step 3: Enter the filename, index block.
Step 4: Print the file name index loop.
Step 5: File is allocated to the unused index blocks
Step 6: This is allocated to the unused linked allocation.
Step 7: Stop the execution

10. Page Replacement Algorithms a. FIFO b. LRU

Objective:

To write a C program to implement page replacement FIFO (First In First Out) algorithm LRU (Least Recently Used)

Procedure:(FIFO)

Step 1: Start the Program
Step 2:Obtain the required data through char and in datatypes.
Step 3:Enter the filename ,index block.
Step 4: Print the file name index loop.
Step 5:Fill is allocated to the unused index blocks
Step 6: This is allocated to the unused linked allocation.
Step 7: Stop the execution

Procedure:(LRU)

Step 1: Start the Program
Step 2:Obtain the required data through char and int datatypes.
Step 3:Enter the filen ame, index block.
Step 4: Print the file name index loop.
Step 5:Fill is allocated to the unused index blocks
Step 6: This is allocated to the unused linked allocation.
Step 7: Stop the execution

11. Simulate Address Translation in Paging

Objective:

To simulate the address translation from logical to physical address under paging

Procedure:

1. Get the range of physical and logical addresses.
2. Get the page size.
3. Get the page number of the data.
4. Construct page table by mapping logical address to physical address.
5. Search page number in page table and locate the base address.
6. Calculate the physical address of the data.

12. Disk Scheduling Techniques

Objective:

Simulation of disk scheduling techniques.

Procedure:

1. Implement scheduling algorithms listed below:
    - **First-in-First-Out (FIFO)**
    - **Shortest Seek Time First (SSTF)**
    - **SCAN**
    - **CSCAN**
    - **LOOK**

2. Calculation of Seek time, transfer time etc.
3. Print the result

Thank You

```
echo"Enter File Name:"
read file
lc=$(wc --lines $file)
wc=$(wc --words $file)
vow=$(grep -o -i" [^AEIOUaeiou]" | wc --words $file)
echo "Lines" $lc
echo "Words" $wc
echo "Vowels Words" $vow**
```

EX No. 3 CPU Scheduling

Dr S.Rajarajan
SASTRA

Objective

Develop programs to demonstrate the various CPU scheduling algorithms and compare their performances through sample sets.

CPU Scheduling

First in first out

Also known as ***F**irst **C**ome, **F**irst **S**erved* (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

- Take the number of processes as input.
- For each process get the arrival time and service time as input.
- Run the following steps until all the processes are completed
- Choose the process that arrived first and has not yet completed
- Update its completion time, mark it as completed, and reduce the number of processes to be completed, update current time

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- Take the number processes as input.
- For each process get the arrival time and service time as input.
- Take the RR time quantum as input.
- Run the following steps until all the processes are completed
- Choose the first process arrived which is not yet completed, give it the fixed RR time quantum
- Reduce the burst time of the process by the RR quantum. If the burst time has hit zero then mark it as completed and update its completion time.

Shortest Job First

This is a non-preemptive approach. The process with the lowest burst time among the arrived processes is chosen

- Take the number of processes as input.
- For each process get the arrival time and service time as input.
- Run the following steps until all the processes are completed
- Choose the process with the smallest burst time among the processes that are not yet completed and have already arrived
- Update its completion time, mark it as completed, reduce the number of processes to be completed and update current time

Shortest Remaining Time

This is a non-preemptive version of the shortest job first algorithm. Initially the process with the lowest service time is taken for execution. But while processing that process if a better process arrives (process with lesser service time than the remaining service time of the currently running process) then immediately preempt or stop the process and start the newly arrived process.

- Take the number processes as input.
- For each process get the arrival time and service time as input.
- In each iteration choose the process with the smallest service time among the not yet completed processes and complete it.
- Keep looking for the arrival of smaller processes.
- Before the current process is completed if a smaller process arrives then halt the execution of the current process and start the execution of new process. Note that for this process you must compare only the remaining service time of the executing process with the new process's service time not the actual service time.
- Reduce the service time of process as service time – time executed.
- Repeat the steps until all the processes are completed.

Sample Code for Priority Scheduling

```c
#include <stdio.h>
struct process
{
    int at;  // arrival time
    int st;  // service time
    int pr;  // priority
    int status; // 1- completed, 0 - not completed
    int ft; // finish time
}ready_list[10];
int n;
int main()
{
    int i,cur_time,pid;
    printf("Enter number of processes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("***********\n");
        printf("Enter Arrival Time:");
        scanf("%d",&ready_list[i].at);
        printf("Enter Burst Time:");
        scanf("%d",&ready_list[i].bt);
        printf("Priority (1-10):");
        scanf("%d",&ready_list[i].pr);
        ready_list[i].status=0;
    }
    i=0; cur_time=0;
    while(i < n)   // until all the processes are completed
    {
        pid=dispatcher(cur_time); // choose the next process for execut
        ready_list[pid].ft=cur_time + ready_list[pid].bt; // update the
        ready_list[pid].status=1;   // Process completed
        cur_time+= ready_list[pid].bt;  // Update clock time
        i++;
    }
    printf("Process\t Arrival Time\t Service Time\t Finish Time\n");
    printf("*******\t ************\t ************\t ***********\n");
    for(i=0;i<n;i++)
```

```
    {
        printf("%d\t\t%d\t\t%d\t\t\t%d\n",i,ready_list[i].at,ready_list
    }
}
int dispatcher(int time)
{
    int i,high_pr=0,index=-1;
    for(i=0;i<n;i++)
    {
        if(ready_list[i].status != 1)   // Process already completed or
        if(ready_list[i].at <= time) // Process's AT is within the curr
        if(ready_list[i].pr > high_pr)  // Choosing high priority proce
        {
            high_pr = ready_list[i].pr;
            index=i;
        }
    }
    return index;
}
```

EX No. 3 CPU Scheduling

Dr S.Rajarajan
SASTRA

Objective

Develop programs to demonstrate the various CPU scheduling algorithms and compare their performances through sample sets.

CPU Scheduling

First in first out

Also known as ***F**irst **C**ome, **F**irst **S**erved* (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

- Take the number of processes as input.
- For each process get the arrival time and service time as input.
- Run the following steps until all the processes are completed
- Choose the process that arrived first and has not yet completed
- Update its completion time, mark it as completed, and reduce the number of processes to be completed, update current time

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- Take the number processes as input.
- For each process get the arrival time and service time as input.
- Take the RR time quantum as input.
- Run the following steps until all the processes are completed
- Choose the first process arrived which is not yet completed, give it the fixed RR time quantum
- Reduce the burst time of the process by the RR quantum. If the burst time has hit zero then mark it as completed and update its completion time.

Shortest Job First

This is a non-preemptive approach. The process with the lowest burst time among the arrived processes is chosen

- Take the number of processes as input.
- For each process get the arrival time and service time as input.
- Run the following steps until all the processes are completed
- Choose the process with the smallest burst time among the processes that are not yet completed and have already arrived
- Update its completion time, mark it as completed, reduce the number of processes to be completed and update current time

Shortest Remaining Time

This is a non-preemptive version of the shortest job first algorithm. Initially the process with the lowest service time is taken for execution. But while processing that process if a better process arrives (process with lesser service time than the remaining service time of the currently running process) then immediately preempt or stop the process and start the newly arrived process.

- Take the number processes as input.
- For each process get the arrival time and service time as input.
- In each iteration choose the process with the smallest service time among the not yet completed processes and complete it.
- Keep looking for the arrival of smaller processes.
- Before the current process is completed if a smaller process arrives then halt the execution of the current process and start the execution of new process. Note that for this process you must compare only the remaining service time of the executing process with the new process's service time not the actual service time.
- Reduce the service time of process as service time – time executed.
- Repeat the steps until all the processes are completed.

Sample Code for Priority Scheduling

```c
#include <stdio.h>
struct process
{
    int at;  // arrival time
    int st;  // service time
    int pr;  // priority
    int status; // 1- completed, 0 - not completed
    int ft; // finish time
}ready_list[10];
int n;
int main()
{
    int i,cur_time,pid;
    printf("Enter number of processes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("***********\n");
        printf("Enter Arrival Time:");
        scanf("%d",&ready_list[i].at);
        printf("Enter Burst Time:");
        scanf("%d",&ready_list[i].bt);
        printf("Priority (1-10):");
        scanf("%d",&ready_list[i].pr);
        ready_list[i].status=0;
    }
    i=0; cur_time=0;
    while(i < n)   // until all the processes are completed
    {
        pid=dispatcher(cur_time); // choose the next process for execut
        ready_list[pid].ft=cur_time + ready_list[pid].bt; // update the
        ready_list[pid].status=1;   // Process completed
        cur_time+= ready_list[pid].bt;  // Update clock time
        i++;
    }
    printf("Process\t Arrival Time\t Service Time\t Finish Time\n");
    printf("*******\t ************\t ************\t ***********\n");
    for(i=0;i<n;i++)
```

```
    {
        printf("%d\t\t%d\t\t%d\t\t\t%d\n",i,ready_list[i].at,ready_list
    }
}
int dispatcher(int time)
{
    int i,high_pr=0,index=-1;
    for(i=0;i<n;i++)
    {
        if(ready_list[i].status != 1)   // Process already completed or
        if(ready_list[i].at <= time) // Process's AT is within the curr
        if(ready_list[i].pr > high_pr)  // Choosing high priority proce
        {
            high_pr = ready_list[i].pr;
            index=i;
        }
    }
    return index;
}
```

Program No. 7

Sequential file allocation

```
#include<stdio.h>
#include<string.h>
void main()
{
int nf=0,i=0,j=0,ch;
char mdname[10],fname[10][10],name[10];
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();
}
```

b. Linked file allocation method

```
#include<stdio.h>
void main()
{
int f[50], p,i, st, len, j, c, k, a;
for(i=0;i<50;i++)
f[i]=0;
printf("Enter how many blocks already allocated: ");
scanf("%d",&p);
printf("Enter blocks already allocated: ");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
x: printf("Enter index starting block and length: ");
scanf("%d%d", &st,&len);
k=len;
if(f[st]==0)
{
for(j=st;j<(st+k);j++)
{
if(f[j]==0)
{
```

```
f[j]=1;
printf("%d-------->%d\n",j,f[j]);
}
else
{
printf("%d Block is already allocated \n",j);
k++;
}
}
}
else
printf("%d starting block is already allocated \n",st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
getch();
}
```

c. indexed file allocation

```
#include<stdio.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
```

```
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
getch();
}
```

PAGE REPLACEMENT Algorithms

a. FIFO

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0,hit,ratio,nr;
label:
printf("\n ENTER THE NUMBER OF PAGES:\n");
scanf("%d",&n);
if(n<=0)
{
printf("\n Enter positive value..!\n\n");

goto label;
}
else
{
printf("\n ENTER THE PAGE NUMBER :\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n ENTER THE NUMBER OF FRAMES :");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]= -1;
j=0;
printf("Ref string\t page frames\n");
for(i=1;i<=n;i++)
{
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
if(frame[k]==a[i])
avail=1;
if (avail==0)
{

frame[j]=a[i];
j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf(" %d ",frame[k]);

}
printf("\n");
}
hit=n-count;

nr=hit*100;
ratio=nr/n;
printf("Page Fault Is %d",count);
printf("\nPage hit is %d",hit);
printf("\nHit Ratio is : %d",ratio);
```

return 0;
}
}

B. Least Recently Used Page replacement algorithm

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

int checkHit(int incomingPage, int queue[], int occupied){

   for(int i = 0; i < occupied; i++){
      if(incomingPage == queue[i])
         return 1;
   }

   return 0;
}

void printFrame(int queue[], int occupied)
{
   for(int i = 0; i < occupied; i++)
      printf("%d\t\t\t",queue[i]);
}

int main()
{

//    int incomingStream[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1};
//    int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3, 6, 1, 2, 4, 3};
   int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3};

   int n = sizeof(incomingStream)/sizeof(incomingStream[0]);
   int frames = 3;
   int queue[n];
   int distance[n];
   int occupied = 0;
   int pagefault = 0;

   printf("Page\t Frame1 \t Frame2 \t Frame3\n");

   for(int i = 0;i < n; i++)
   {
      printf("%d:  \t\t",incomingStream[i]);
      // what if currently in frame 7
      // next item that appears also 7
      // didnt write condition for HIT

      if(checkHit(incomingStream[i], queue, occupied)){
         printFrame(queue, occupied);
      }

      // filling when frame(s) is/are empty
      else if(occupied < frames){
         queue[occupied] = incomingStream[i];
```

```
            pagefault++;
            occupied++;

            printFrame(queue, occupied);
        }
        else{

            int max = INT_MIN;
            int index;
            // get LRU distance for each item in frame
            for (int j = 0; j < frames; j++)
            {
                distance[j] = 0;
                // traverse in reverse direction to find
                // at what distance  frame item occurred last
                for(int k = i - 1; k >= 0; k--)
                {
                    ++distance[j];

                    if(queue[j] == incomingStream[k])
                        break;
                }

                // find frame item with max distance for LRU
                // also notes the index of frame item in queue
                // which appears furthest(max distance)
                if(distance[j] > max){
                    max = distance[j];
                    index = j;
                }
            }
            queue[index] = incomingStream[i];
            printFrame(queue, occupied);
            pagefault++;
        }

        printf("\n");
    }

    printf("Page Fault: %d",pagefault);

    return 0;
}
```

O

a. Producer consumer problem using semaphores

```
#include <stdio.h>
int empty=5,full=0,s=1;
void wait(int* s){
   while(*s<=0);
   (*s)--;
}
void signal(int* s){
   (*s)++;
}
void producer(int buffer[]){
   if(empty<=0){
      printf("buffer is full");
      return;
   }
   wait(&empty);
   wait(&s);
   printf("Enter a item to produce");
   scanf("%d",&buffer[full]);
   signal(&s);
   signal(&full);
}
void consumer(int buffer[]){
   if(full<=0){
      printf("buffer is empty\n");
      return;
   }
   wait(&full);
   wait(&s);
   printf("%d is consumed.",buffer[full]);
   signal(&s);
   signal(&empty);
}
int main() {
   int a[5],ch;
   do{

      printf("\n1.Produce\n2.Consume\n3.Exit\nEnter your choice: ");
      scanf("%d",&ch);
      switch(ch){
         case 1:
         producer(a);
         break;
         case 2:
         consumer(a);
         break;
         default:
         printf("Enter valid choice!!!");
         break;
      }
   }while(ch!=3);

   return 0;
```

```
}
```

2. Solution for Dining philosophers problem

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

sem_t room;
sem_t chopstick[5];

void * philosopher(void *);
void eat(int);
int main()
{
	int i,a[5];
	pthread_t tid[5];

	sem_init(&room,0,4);

	for(i=0;i<5;i++)
		sem_init(&chopstick[i],0,1);

	for(i=0;i<5;i++){
		a[i]=i;
		pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
	}
	for(i=0;i<5;i++)
		pthread_join(tid[i],NULL);
}

void * philosopher(void * num)
{
	int phil=*(int *)num;

	sem_wait(&room);
	printf("\nPhilosopher %d has entered room",phil);
	sem_wait(&chopstick[phil]);
	sem_wait(&chopstick[(phil+1)%5]);

	eat(phil);
	sleep(2);
	printf("\nPhilosopher %d has finished eating",phil);

	sem_post(&chopstick[(phil+1)%5]);
```

```
        sem_post(&chopstick[phil]);
        sem_post(&room);
}

void eat(int phil)
{
        printf("\nPhilosopher %d is eating",phil);
}
```

```
//EXP9  Single level directory
#include<stdio.h>
#include<string.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;
int main()
{
int i,ch;
char f[30];

dir.fcnt = 0;
printf("\nEnter name of directory -- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your choice
-- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n Enter the name of the file -- ");
scanf("%s",dir.fname[dir.fcnt]);
dir.fcnt++;
break;
case 2: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
```

```
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is deleted ",f);
strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
else
dir.fcnt--;
break;
case 3: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is found ", f);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
break;
case 4: if(dir.fcnt==0)
printf("\n Directory Empty");
else
{
printf("\n The Files are -- ");
for(i=0;i<dir.fcnt;i++)
```

```
printf("\t%s",dir.fname[i]);
}
break;
default: return 0;
}
}
return 0;
}
```

```
//exp9:TWO Level DIRECTORY
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir[10];
int
 main()
{
int i,ch,dcnt,k;
char f[30], d[30];
dcnt=0;
while(1)
{
printf("\n\n 1. Create Directory\t 2. Create File\t 3. Delete File");
printf("\n 4. Search File \t \t 5. Display \t 6. Exit \t Enter your choice -- ");
scanf("%d",&ch);
switch(ch)
```

```
{
case 1: printf("\n Enter name of directory -- ");
scanf("%s", dir[dcnt].dname);
dir[dcnt].fcnt=0;
dcnt++;
printf("Directory created");
break;
case 2: printf("\n Enter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",dir[i].fname[dir[i].fcnt]);
dir[i].fcnt++;
printf("File created");
break;
}
if(i==dcnt)
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
```

```
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f);
dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{

if(strcmp(d,dir[i].dname)==0)
{
printf("Enter the name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is found ",f);
goto jmp1;
}
}
```

```
printf("File %s not found",f);
goto jmp1;
}
}
printf("Directory %s not found",d);
jmp1: break;
case 5: if(dcnt==0)
printf("\nNo Directory's ");
else
{
printf("\nDirectory\tFiles");
for(i=0;i<dcnt;i++)
{
printf("\n%s\t\t",dir[i].dname);
for(k=0;k<dir[i].fcnt;k++)
printf("\t%s",dir[i].fname[k]);
}
}
break;
default:exit(1);
}
}
return 0;
}
```

Exercise 11

Program: ADDRESS TRANSLATION

```
#include<stdio.h>
int main()
{
   int pageno,page[10]={10,20,30,40,50},frameno[10],i,flag=0;
   printf("\n\tpage number\tframenumber");
   for(i=0;i<5;i++)
   {
      frameno[i]=page[i];
      printf("\n\t%d\t\t\t%d",page[i],i);
   }
   printf("\nEnter the page number: ");
   scanf("%d",&pageno);
   for(i=0;i<5;i++)
   {
      if(pageno==frameno[i])
      {
         printf("\nThe Page %d is found",pageno);
         flag=1;
         break;
      } }
      if(flag==0)
      {
          printf("\nPage fault");
      }
   return 0;

}
```

DISC SCHEDULING (FCFS):

```
#include<stdio.h>
int main()
{
   int n,a[10],head,totaltime,i;
   printf("\nEnter the n value: "); //Get n value
   scanf("%d",&n);
   printf("\nEnter %d n values",n); //read the input from user
   for(i=1;i<=n;i++)
   {
      scanf("%d",&a[i]); //To get multiple input from user we use iterate/looping statement
   }
   printf("\nEnter the head value: "); //Get head value
   scanf("%d",&head);
   for(i=1;i<=n;i++)
   {
   totaltime=totaltime+abs(head-a[i]);    //Formula to find/calculate total time
   head=a[i];
}
printf("The total time is %d",totaltime); //Now we get the result of calculation of total time.
return 0;
}
```

DISK SCHEDULING ALGORITHM-SCAN

CODE:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
   int a[50],n,head,right,left,i,j,d,val,temp;
   printf("\nenter the number of disk request");
   scanf("%d",&n);
   printf("\nenter the disk request");
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nenter the head value");
   scanf("%d",&head);
   printf("\nenter the starting value");
   scanf("%d",&left);
   printf("\nenter the end value");
   scanf("%d",&right);
   printf("\nenter the direction to move\npress 0 to left\tpress 1 to right ");
   scanf("%d",&d);
   //sorting
   printf("\nsorted elements");
```

```
    for(i=0;i<n;i++)
    {
       for(j=i+1;j<n;j++)
       {
          if(a[i]>a[j])
          {
             temp=a[i];
             a[i]=a[j];
             a[j]=temp;
          }
       }
    }
    if(d==0)
    {
       val=(head-left)+(left+a[n-1]);
       printf("\naccess time=%d",val);
    }
    else if(d==1)
    {
       val=abs(head-right)+(right-a[0]);
       printf("\naccess time=%d",val);
    }
}
```

DISK SCHEDULING ALGORITHM - CSCAN

CODE:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int a[50],n,head,right,left,i,j,k=-1,d,val,temp,templeft[50],temprt[50];
    printf("\nenter the number of disk request");
    scanf("%d",&n);
    printf("\nenter the disk request");
    for(i=0;i<n;i++)
    {
       scanf("%d",&a[i]);
    }
    printf("\nenter the starting value");
    scanf("%d",&left);
    printf("\nenter the end value");
    scanf("%d",&right);
    printf("\nenter the head value");
    scanf("%d",&head);
    printf("\nenter the direction to move\npress 0 to left\tpress 1 to right ");
    scanf("%d",&d);
    //sorting
       for(i=0;i<n;i++)
    {
```

```
    for(j=i+1;j<n;j++)
    {
      if(a[i]>a[j])
      {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
      }
    }
  }
  j=0;
  //k=0;
  for(i=0;i<n;i++)
  {
    if(a[i]<head)
    {
      templeft[j]=a[i];
      j++;
    }
    else if(a[i]>head)
    {
      k++;
      temprt[k]=a[i];
      //c++;
    }
  }

  printf("\nsorted elements are");
  for(i=0;i<j;i++)
  {
    printf("\t%d",templeft[i]);
  }
  if(d==0)
  {
    val=(head-left)+abs(left-right)+(right-temprt[0]);
    printf("\naccess time=%d",val);
  }
  else if(d==1)
  {
    val=abs(head-right)+abs(right-left)+abs(left-templeft[j-1]);
    printf("\naccess time=%d",val);
  }
}
```

DISK SCHEDULING ALGORITHM-LOOK
CODE:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
  int a[50],n,head,right,left,i,j,k=-1,d,val,temp,templeft[50],temprt[50];
```

```
printf("\nenter the number of disk request");
scanf("%d",&n);
printf("\nenter the disk request");
for(i=0;i<n;i++)
{
   scanf("%d",&a[i]);
}
printf("\nenter the starting value");
scanf("%d",&left);
printf("\nenter the end value");
scanf("%d",&right);
printf("\nenter the head value");
scanf("%d",&head);
printf("\nenter the direction to move\npress 0 to left\tpress 1 to right ");
scanf("%d",&d);
//sorting
   for(i=0;i<n;i++)
{
   for(j=i+1;j<n;j++)
   {
      if(a[i]>a[j])
      {
         temp=a[i];
         a[i]=a[j];
         a[j]=temp;
      }
   }
}
j=0;
//k=0;
for(i=0;i<n;i++)
{
   if(a[i]<head)
   {
      templeft[j]=a[i];
      j++;
   }
   else if(a[i]>head)
   {
      k++;
      temprt[k]=a[i];
      //c++;
   }
}

printf("\nsorted elements are");
for(i=0;i<j;i++)
{
   printf("\t%d",templeft[i]);
}
if(d==0)
{
   val=abs(head-templeft[0])+abs(templeft[0]-temprt[k]);
```

```
        printf("\naccess time=%d",val);
    }
    else if(d==1)
    {
        val=abs(head-temprt[k])+abs(temprt[k]-templeft[0]);
        printf("\naccess time=%d",val);
    }
}
```