KLS
Vishwanathrao Deshpande Institute of Technology
Haliyal – 581329

OPERATING SYSTEM LABORATORY MANUAL

[BCS303]
for
III Semester B.E.

As prescribed by

VISVESVARAYA TECHNOLOGICAL UNIVERSITY,
BELAGAVI – 590014
(From Academic Year: 2025-26)

Prepared by

Prof. Meenakshi Nadagouda

Department of Computer Science and Engineering (AIML)

Index

PROGRAM OUTCOMES(POs)

Program Outcomes as defined by NBA (PO) Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

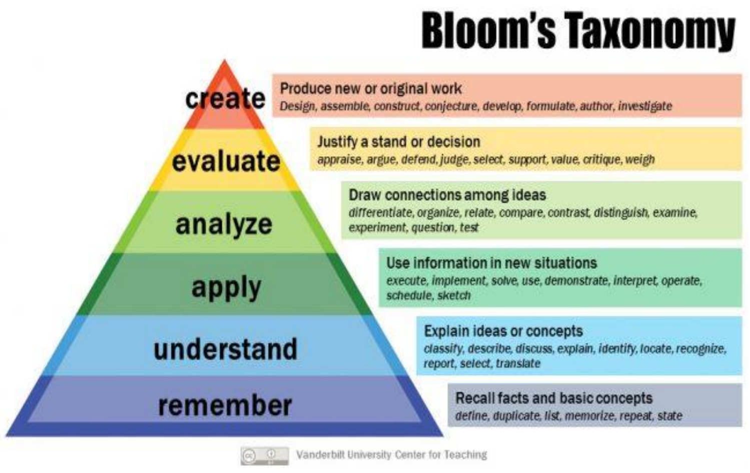
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.


Bloom's Taxonomy
create
Produce new or original work
Design, assemble, construct, conjecture, develop, formulate, author, investigate
evaluate
Justify a stand or decision
appraise, argue, defend, judge, select, support, value, critique, weigh
analyze
Draw connections among ideas
differentiate, organize, relate, compare, contrast, distinguish, examine, experiment, question, test
apply
Use information in new situations
execute, implement, solve, use, demonstrate, interpret, operate, schedule, sketch
understand
Explain ideas or concepts
classify, describe, discuss, explain, identify, locate, recognize, report, select, translate
remember
Recall facts and basic concepts
define, duplicate, list, memorize, repeat, state
Vanderbilt University Center for Teaching

Vision (College)	
To nurture talent and enrich society through excellence in technical education, research and innovation.	
Mission (College)	
1.To augment innovative Pedagogy, kindle quest for interdisciplinary learning & to enhance conceptual understanding. 2.To build competence, professional ethics & develop entrepreneurial thinking. 3.To strengthen Industry Institute Partnership & explore global collaborations. 4.To inculcate culture of socially responsible citizenship. 5.To focus on Holistic & Sustainable development.	
Vision (Dept)	
To achieve excellence in technical education, research, and innovation in computer science and Engineering by emphasizing on global trending technologies.	
Mission (Dept)	
1. To train students with conceptual understanding through innovative pedagogies. 2. To imbibe professional, research, and entrepreneurial skills with a commitment to the nation's development at large. 3. To strengthen the industry-institute Interaction. 4. To promote life–long learning with a sense of societal & ethical responsibilities.	
Program Educational Objectives (PEO)	
PEO1	To develop an ability to identify and analyze the requirements of Computer Science and Engineering in design and providing novel engineering solutions.
PEO2	To develop abilities to work in team on multidisciplinary projects with effective communication skills, ethical qualities and leadership roles.
PEO3	To develop abilities for successful Computer Science Engineer and achieve higher career goals.
Program Specific Outcomes (PSO)	
PSO 1	To develop the ability to model real-world problems using appropriate data structure and suitable algorithms in the area of Data Processing, System Engineering, and Networking for varying complexity.
PSO 2	To develop an ability to use modern computer languages, environments and platforms in creating innovative career.

Lab Program 1.

Develop a C program to implement the process system calls (fork(), wait() and exec() create process, terminate process).

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
        pid_t pid;
        pid =fork();                    /* fork a child process */
        if (pid < 0)
        {
        fprintf(stderr, "Fork Failed"); /*error occurred */
        return 1;
        }
        else if (pid == 0)                      /*child process */
        {
        execlp("/bin/ls","ls",NULL);
        printf ( "Child: pid value = %d\n" , pid);
        printf ( "Child: Process id = %d\n" , getpid());
        printf ( "Child: Parent-Process id = %d\n" , getppid());
        printf ( "Child: Exiting from child\n" );
        exit (0);
        }
        else
        {                               /* parent process */
                                        /*parent will wait for the child to complete*/
        wait (NULL) ;
        printf("Child Complete");
        printf ( "Parent: Waiting for child\n" );
        wait(0);
        printf ( "Parent: pid value = %d\n" , pid);
        printf ( "Parent: Process id = %d\n" , getpid());
        printf ( "Parent: Child-Process id = %d\n" , pid);
        }
        return 0;
}
```

```
n@ubuntu-VirtualBox:~/Documents/OS/os2$ gcc fork.c
n@ubuntu-VirtualBox:~/Documents/OS/os2$ ./a.out
Parent: Waiting for child
Child: pid value = 0
Child: Process id = 3792
Child: Parent-Process id = 3791
Child: Exiting from child
Parent: pid value = 3792
Parent: Process id = 3791
Parent: Child-Process id = 3792
```

Exec.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<wait.h>
int main ( int argc, char * argv[])
{
if (argc != 3)
{
printf ( "Usage: execl <int1> <int2>\n" );
exit (0);
}
pid_t pid = fork();

if (pid < 0)
{
printf ( "Fork failed\n" );
exit (0);
}
else if (pid == 0)
{
printf ( "Child:\n" );
execl( "sum_diff" , "sum_diff" , argv[1], argv[2], NULL );
exit (0);
}
else
{
wait(0);
printf ( "Parent:\n" );
}

}
```

```
n@ubuntu-VirtualBox:~/Documents/OS/os2$ gcc sum_diff.c -o sum_diff
n@ubuntu-VirtualBox:~/Documents/OS/os2$ gcc execl.c
n@ubuntu-VirtualBox:~/Documents/OS/os2$ ./a.out 5 5
Child:
Sum of 5 and 5 is 10
Difference of 5 and 5 is 0
Parent:
```

Lab Program 2

Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a). FIRST COME FIRST SERVE:

AIM: To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process name and the burst time Step
4: Set the waiting of the first process as ‗
0‘and its burst time as its turnaround time Step
5: for each process in the Ready Q calculate
a). Waiting time (n) = waiting time (n-1) + Burst time
(n-1) b).
Turnaround time (n)= waiting time(n)+Burst time(n)
Step 6: Calculate a) Average waiting time = Total waiting Time / Number of process
b) Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop the process

Source Code

```
#include<stdio.h>
#include<conio.h>
main()
{
int
bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
        printf("\nEnter Burst Time for Process %d -- ", i);
        scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
```

```
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

INPUT

```
Enter the number of processes -- 3
Enter Burst Time for Process 0 -- 24
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 – 3
```

OUTPUT

```
PROCESS          BURST TIME        WAITING TIME       TURNAROUND TIME
P0               24                0                  24
P1               3                 24                 27
P2               3                 27                 30

Average Waiting Time-- 17.000000
Average Turnaround Time -- 27.000000
```

b). SHORTEST JOB FIRST:

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

DESCRIPTION:

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

```
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as _0‘ and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burst time
Step 7: For each process in the ready queue, calculate
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8: Calculate c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time/ Number of process
Step 9: Stop the process
```

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}

for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
        temp=bt[i];
        bt[i]=bt[k];
        bt[k]=temp;
        temp=p[i];
        p[i]=p[k];
        p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
        printf("\n\t P%d \t\t %d \t\t %d\t\t %d", p[i], bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n);
        printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();

}
```

INPUT

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 6
Enter Burst Time for Process 1 -- 8
Enter Burst Time for Process 2 -- 7
Enter Burst Time for Process 3 -- 3
```

OUTPUT

```
PROCESS        BURST TIME     WAITING TIME     TURNAROUND TIME
P3             3              0                3
P0             6              3                9
P2             7              9                16
P1             8              16               24

Average Waiting Time            -- 7.000000
Average Turnaround Time         -- 13.000000
```

c). ROUND ROBIN:

AIM: To simulate the CPU scheduling algorithm round-robin.

DESCRIPTION:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice
Step 5: If the burst time is less than the time slice then the no. of time slices =1.
Step 6: Consider the ready queue is a circular Q, calculate
a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).
Step 7: Calculate
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process Step
8: Stop the process

SOURCE CODE

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float
awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else
{
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++){
wa[i]=tat[i]-ct[i];
att+=tat[i];
awt+=wa[i];}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME\t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
}
```

INPUT:

```
Enter the no of processes – 3
Enter Burst Time for process 1 – 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 – 3
Enter the size of time slice – 3
```

OUTPUT:

```
PROCESS      BURST TIME          WAITING TIME        TURNAROUNDTIME
1            24                  6                   30
2            3                   4                   7
3            3                   7                   10

The Average Turnaround time is – 15.666667
The Average Waiting time is ------------ 5.666667
```

d). PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

```
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0‘and its burst time as its turnaround time
Step 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Q calculate
Step 8: for each process in the Ready Q calculate
        a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
        b) Turnaround time (n)= waiting time(n)+Burst time(n)
Step 9: Calculate
        c) Average waiting time = Total waiting Time / Number of process
        d) Average Turnaround time = Total Turnaround Time / Number of process
        Print the results in an order.
Step10: Stop
```

SOURCE CODE:

```
#include<stdio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i);
scanf("%d %d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k])
{
temp=
p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n); printf("\nAverage
Turnaround Time is --- %f",tatavg/n);
getch();
}
```

INPUT

```
Enter the number of processes -- 5
Enter the Burst Time & Priority of Process 0 --- 10           3
Enter the Burst Time & Priority of Process 1 --- 1            1
Enter the Burst Time & Priority of Process 2 --- 2            4
Enter the Burst Time & Priority of Process 3 --- 1            5
Enter the Burst Time & Priority of Process 4 --- 5            2
```

OUTPUT

```
PROCESS     PRIORITY    BURST TIME  WAITING TIME        TURNAROUND TIME
1           1           1           0                   1
4           2           5           1                   6
0           3           10          6                   16
2           4           2           16                  18
3           5           1           18                  19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000
```

VIVA QUESTIONS

1) Define the following
a) Turnaround time b) Waiting time c) Burst time d) Arrival time
2) What is meant by process scheduling?
3) What are the various states of process?
4) What is the difference between preemptive and non-preemptive scheduling
5) What is meant by time slice?
6) What is round robin scheduling?

Lab program 3: Develop a C program to simulate producer-consumer problem using semaphores.

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#define MAX 5
sem_t in_sem;
sem_t rm_sem;
void * producer ()

{
int count = 0;
while (1)
{
sem_wait(&in_sem);
sleep(rand()%3);
printf ( "Producer inserted at %d\n" , count % MAX + 1);
count++;
sem_post(&rm_sem);
}
}
void * consumer ()

{
int count = 0;
while (1)
{
sem_wait(&rm_sem);
printf ( "Consumer removed at %d\n" , count % MAX + 1);
sleep(rand()%3);
count++;
sem_post(&in_sem);
}
}

int main ()
{
pthread_t threads[2];
sem_init(&in_sem, 0, MAX);
sem_init(&rm_sem, 0, 0);
srand(time( NULL ));
pthread_create(&threads[0], NULL , producer, NULL );
pthread_create(&threads[1], NULL , consumer, NULL );
pthread_join(threads[0], NULL );
pthread_join(threads[1], NULL );
sem_destroy(&in_sem);
sem_destroy(&rm_sem);
}
```

```
n@ubuntu-VirtualBox:~/Documents/OS/os5$ gcc producer_consumer.c -l
pthread
n@ubuntu-VirtualBox:~/Documents/OS/os5$ ./a.out
Producer inserted at 1
Consumer removed at 1
Producer inserted at 2
Producer inserted at 3
Consumer removed at 2
Consumer removed at 3
Producer inserted at 4
Consumer removed at 4
Producer inserted at 5
Producer inserted at 1
Consumer removed at 5
Producer inserted at 2
Producer inserted at 3
Consumer removed at 1
Producer inserted at 4
Producer inserted at 5
Consumer removed at 2
Consumer removed at 3
Consumer removed at 4
Consumer removed at 5
Producer inserted at 1
Consumer removed at 1
Producer inserted at 2
Consumer removed at 2
Producer inserted at 3
Producer inserted at 4
Producer inserted at 5
Consumer removed at 3
Consumer removed at 4
Producer inserted at 1
```

Lab program 4:

Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close API's in your program.

Reader_writer.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#define N 10
int process[N] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int r_count = 0;
sem_t rw_sem;
sem_t rc_sem;

void * reader ( void * num)
{
int n = *( int *)num;
sem_wait(&rc_sem);
if (!r_count)
{
printf ( "Waiting for reading\n" );
sem_wait(&rw_sem);
}
r_count++;
sem_post(&rc_sem);
printf ( "Process %d is reading\n" , n+1);
sleep(2);

sem_wait(&rc_sem);
r_count--;
if (!r_count)
{
printf ( "Finished reading\n" );
sem_post(&rw_sem);
}
sem_post(&rc_sem);
}

void * writer ( void * num)
{
int n = *( int *)num;
printf ( "Waiting for writing\n" );
sem_wait(&rw_sem);
printf ( "Process %d is writing\n" , n+1);
sleep(4);
printf ( "Finished writing\n" );
sem_post(&rw_sem);
}
```

```
int main ()
{
pthread_t threads[N];
sem_init(&rw_sem, 0, 1);
sem_init(&rc_sem, 0, 1);
srand(time( NULL ));
int r = rand()%N;
for ( int i = 0; i < N; i++)
{
if (r == i)
{
pthread_create(&threads[i], NULL , writer, &process[i]);
r = rand()%(N-i) + i;
continue ;
}

pthread_create(&threads[i], NULL , reader, &process[i]);
}

for ( int i = 0; i < N; i++)
{
pthread_join(threads[i], NULL );
}
sem_destroy(&rw_sem);
sem_destroy(&rc_sem);
}
```

```
n@ubuntu-VirtualBox:~/Documents/OS/os5$ gcc reader_writer.c -l pthread
n@ubuntu-VirtualBox:~/Documents/OS/os5$ ./a.out
Waiting for reading
Process 3 is reading
Waiting for writing
Process 5 is reading
Process 6 is reading
Process 7 is reading
Process 8 is reading
Process 9 is reading
Waiting for writing
Process 2 is reading
Process 1 is reading
Finished reading
Process 4 is writing
Finished writing
Process 10 is writing
Finished writing
```

Lab program 5: Develop a C program to simulate Bankers Algorithm for Deadlock Avoidance

DESCRIPTION:
Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

n-Number of process,
m-number of resource types.
Available: Available[j]=k, k – instance of resource type Rj is available.
Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]- Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively,
Work=Available and
Finish[i] =False.
2. Find an i such that both
Finish[i] =False
Need<=Work
If no such I exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm
Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources areavailable.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

Available=Available-Request I;
Allocation I=Allocation +Request I;
Need i=Need i- Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.
***11.** end*

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void
main()
{
int alloc[10][10],max[10][10];
int avail[10],work[10],total[10];
int i,j,k,n,need[10][10];
int m;
int
count=0,c=0;
char finish[10];
clrscr();
printf("Enter the no. of processes and resources:");
scanf("%d%d",&n,&m);
for(i=0;i<=n;i++)
finish[i]='n';
printf("Enter the claim matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<m;j++)
scanf("%d",&max[i][j]);
printf("Enter the allocation matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<m;j++)
scanf("%d",&alloc[i][j]);
printf("Resource vector:");
for(i=0;i<m;i++)
scanf("%d",&total[i]);
for(i=0;i<m;i++)
avail[i]=0;
for(i=0;i<n;i++)
for(j=0;j<m;j++)
avail[j]+=alloc[i][j];
for(i=0;i<m;i++)
work[i]=avail[i];
for(j=0;j<m;j++)
work[j]=total[j]-work[j];
for(i=0;i<n;i++)
for(j=0;j<m;j++)
need[i][j]=max[i][j]-alloc[i][j];
A:
for(i=0;i<n;i++)
{
c=0;
for(j=0;j<m;j++)
if((need[i][j]<=work[j])&&(finish[i]=='n'))
c++;

if(c==m)
{
printf("All the resources can be allocated to Process %d", i+1);
printf("\n\nAvailable resources are:");
for(k=0;k<m;k++)
{
work[k]+=alloc[i][k];
printf("%4d",work[k]);
}
printf("\n");
finish[i]='y';
printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
count++;
}
}
if(count!=n)
goto A;
else
printf("\n System is in safe mode");
printf("\n The given state is safe state");
getch();
}
```

OUTPUT

```
Enter the no. of processes and resources: 4 3
Enter the claim matrix:
3 2 2
6 1 3
3 1 4
4 2 2

Enter the allocation matrix:
1 0 0
6 1 2
2 1 1
0 0 2

Resource vector: 9 3 6
All the resources can be allocated to Process 2
Available resources are: 6 2 3
Process 2 executed?:y
All the resources can be allocated to Process 3 Available resources
are: 8 3 4
Process 3 executed?:y
All the resources can be allocated to Process 4 Available resources
are: 8 3 6
Process 4 executed?:y
All the resources can be allocated to Process 1
Available resources are: 9 3 6

Process 1 executed?: y
System is in safe mode
The given state is safe state.
```

VIVA QUESTIONS

1) What is meant by deadlock?
2) What is safe state in banker's algorithms?
3) What is banker's algorithm?
4) What are the necessary conditions where deadlock occurs?
5) What are the principles and goals of protection?

Lab Program 6:
Develop a C Program to simulate the following contiguous memory allocation techniques.

AIM: To Write
a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION
One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.
Best-fit strategy chooses the block that is closest in size to the request.
First-fit chooses the first available block that is large enough.
Worst-fit chooses the largest available block.

WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp; static int bf[max],ff[max];
clrscr( );
printf("\n\tMemory Management Scheme- First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
        {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
        }
printf("Enter the size of the files :-\n");
        for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
        {
                for(j=1;j<=nb;j++)
                        {
                                if(bf[j]!=1)
                                        {
                                                temp=b[j]-f[i];
                                                if(temp>=0)
                                                        {
                                                                ff[i]=j;
                                                                break;
                                                        }
                                        }
                        }
                        frag[i]=temp;
                        bf[ff[i]]=1;
        }
        printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
        for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
        getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:
-
Block 1: 5
Block 2: 2
Block 3: 7
Enter the size of the files:
-
File 1: 1
File 2: 4
```

OUTPUT

```
File No        File Size          Block No         Block Size        Fragment
1              1                  1                5                 4
2              4                  3                7                 3
```

BEST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
clrscr();
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
printf("Block %d:",i);
scanf("%d",&b[i]);
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
                {
                        for(j=1;j<=nb;j++)
                                {
                                        if(bf[j]!=1)
                                        {
                                                temp=b[j]-f[i];
                                                if(temp>=0)
                                                        if(lowest>temp)
                                                        {
                                                                ff[i]=j;
                                                                lowest=temp;
                                                        }
                                        }}
                        frag[i]=lowest; bf[ff[i]]=1; lowest=10000;
}
printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7
```

```
Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
        File No         File Size       Block No        Block Size              Fragment
        1               1               2               2                       1
        2               4               1               5                       1
```

FIRST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int
frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0; static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
for(i=1;i<=nf;i++)
        {
        for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1) //if bf[j] is not allocated
                        {
                                temp=b[j]-f[i];
                                if(temp>=0)
                                        if(highest<temp)
                                        {
                        }
                }
                frag[i]=highest; bf[ff[i]]=1; highest=0;
        }
        ff[i]=j; highest=temp;
}
        printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragement");
        for(i=1;i<=nf;i++)
```

```
                printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
        getch();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No         File Size       Block No        Block Size      Fragment
1               1               3               7               6
2               4               1               5               1
```

Lab Program 7: Develop a C program to simulate page replacement algorithms.

a) FIFO **b) LRU**

DESCRIPTION:
Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.
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. LRU-
In this algorithm page will be replaced which is least recently used.

ALGORITHM:
1. Start the process
2. Read number of pages n
3. Read number of pages no
4. Read page numbers into an array a[i]
5. Initialize avail[i]=0 .to check page hit
6. Replace the page with circular queue, while re-placing check page availability in the frame Place avail[i]=1 if page is placed in the frame Count page faults
7. Print the results.
8. Stop the process.

a) FIRST IN FIRST OUT SOURCE CODE:

```
#include<stdio.h>
#include<conio.h> int fr[3];
void main()
{
void display();
int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};
int flag1=0,flag2=0,pf=0,frsize=3,top=0;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0; flag2=0; for(i=0;i<12;i++)
{
if(fr[i]==page[j])
{
flag1=1; flag2=1; break;
}
}
if(flag1==0)
{
for(i=0;i<frsize;i++)
{
if(fr[i]== -1)
{
fr[i]=page[j]; flag2=1; break;
}
}
}
if(flag2==0)
{
fr[top]=page[j];
top++;
pf++;
if(top>=frsize)
top=0;
}
display();
}

printf("Number of page faults : %d ",pf+frsize);
getch();
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("%d\t",fr[i]);
}
```

OUTPUT:

```
2 -1 -1
2 3 -1
2 3 -1
2 3 1
5 3 1
5 2 1
5 2 4
5 2 4
3 2 4
3 2 4
3 5 4
3 5 2

Number of page faults: 9
```

b) LEAST RECENTLY USED

AIM: To implement LRU page replacement technique.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
int fr[3];
void main()
{
void display();
int p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i, j, fs[3];
int index,k,l,flag1=0,flag2=0,pf=0,frsize=3;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0,flag2=0;
for(i=0;i<3;i++)
{
if(fr[i]==p[j])
{
flag1=1;
flag2=1; break;
}
}
if(flag1==0)
{
for(
i=0;i<3;i++)
{
if(fr[i]== -1)
{
fr[i]=p[j]; flag2=1;
break;
}
}
}
if(flag2==0)
{
for(i=0;i<3;i++)
fs[i]=0;
for (k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<3;i++)
{
if(fr[i]==p[k]) fs[i]=1;
}}
for(i=0;i<3;i++)
{
if(fs[i]==0)
index=i;
}
fr[index]=p[j];
pf++;
}
display();
}
printf("\n no of page faults :%d",pf+frsize);
getch();
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("\t%d",fr[i]);
}
```

OUTPUT:

```
2 -1 -1
2 3 -1
2 3 -1
2 3 1
2 5 1
2 5 1
2 5 4
2 5 4
3 5 4
3 5 2
3 5 2
3 5 2

No of page faults: 7
```

VIVA QUESTIONS

1) What is meant by page fault?
2) What is meant by paging?

3) What is page hit and page fault rate?
4) List the various page replacement algorithm
5) Which one is the best replacement algorithm?

Lab Program 8

Simulate following file organization techniques

a) **Single level directory b) Two level directory**

a) Single level directory

AIM: Program to simulate Single level directory file organization technique.

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.

SOURCE CODE :

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;
void
main()
{
int i,ch; char
f[30]; clrscr();
dir.fcnt = 0;
printf("\nEnter name of directory-- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n\n1. Create File\t2. Delete File\t3. Search File \n4. Display Files\t5. Exit\nEnter your
choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter the name of the file -- ");
        scanf("%s",dir.fname[dir.fcnt]);
        dir.fcnt++;
        break;
case 2: printf("\nEnter 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("\nEnter 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("\nDirectory Empty");
          else
          {
          printf("\nThe Files are -- ");
          for(i=0;i<dir.fcnt;i++)
          printf("\t%s",dir.fname[i]);
          }
break;
default: exit(0);
}
}
getch();}
```

OUTPUT:

```
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
```

b) Two level directory

AIM: Program to simulate two level file organization technique

Description: 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.

SOURCE CODE :

```
#include<stdio.h>
struct
{
char
dname[10],fname[10][10];
int fcnt;
}dir[10];

void main()
{
int i,ch,dcnt,k; char
f[30], d[30]; clrscr();
dcnt=0;
        while(1)
        {
        printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
        printf("\n4. Search File\t\t5. Display\t6. Exit\t
        Enter your choice --");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("\nEnter name of directory-- ");
                  scanf("%s", dir[dcnt].dname);
                  dir[dcnt].fc
                  nt=0;
                  dcnt++;
                  printf("Directory created"); break;
        case 2: 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",dir[i].fname[dir[i].fcnt]);
                  dir[i].fcnt++;
                  printf("File created");
        }
        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++)
                  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(0);
                }
                }
        getch();
}
```

OUTPUT

```
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 – 6
```

VIVA QUESTIONS

1. Define directory?
2. List the different types of directory structures?
3. What is the advantage of hierarchical directory structure?
4. Which of the directory structures is efficient? Why?
5. What is acyclic graph directory?

Lab Program 9: Develop a C program to simulate the Linked file allocation strategies.

AIM:
To implement linked file allocation technique.

DESCRIPTION:
In the chained method file allocation table contains a field which points to starting block of memory. From it for each block a pointer is kept to next successive block. Hence, there is no external fragmentation.

ALGORTHIM:
```
Step 1: Start the program.
Step 2: Get the number of files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations by selecting a location randomly
        q= random(100);
        a) Check whether the selected location is free.
        b) If the location is free allocate and set flag=1 to the allocated locations.
        While allocating next location address, attach it to previous location.
for(i=0;i<n;i++)
{
        for(j=0;j<s[i];j++)
                {
                        q=random(100); if(b[q].flag==0)
                        b[q].flag=1;
                        b[q].fno=j;
                        r[i][j]=q;
                        if(j>0)
                        {
                }
}
p=r[i][j-1]; b[p].next=q;}

Step 5: Print the results file no, length, Blocks allocated.

Step 6: Stop the program
```

SOURCE CODE :

```
#include<stdio.h>
main()
{
        int
        f[50],p,i,j,k,a,st,len,n,c;
        clrscr();
        for(i=0;i<50;i++) f[i]=0;
        printf("Enter how many blocks that are already
        allocated"); scanf("%d",&p);
        printf("\nEnter the blocks no.s that are already allocated");
        for(i=0;i<p;i++)
        {
                scanf("%d",&a);
                f[a]=1;
        }
        X:
        printf("Enter the starting index block &
        length"); scanf("%d%d",&st,&len); k=len;
        for(j=st;j<(k+st);j++)
        {
                if(f[j]==0)
                {
                        f[j]=1;
                        printf("\n%d->%d",j,f[j]);
                }
                else
                {
                        printf("\n %d->file is already allocated",j);
                        k++;
                }
        }
        printf("\n If u want to enter onemore file? (yes-1/no-0)");
        scanf("%d",&c);
                if(c==1)
                goto X;
                else
                exit();
        getch( );
}
```

OUTPUT:

```
Enter how many blocks that are already allocated 3
Enter the blocks numbers that are already allocated 4 7
Enter the starting index block & length 3 7 9
3->1
4->1 file is already allocated
5->1
6->1
7->1 file is already allocated
8->1
9->1file is already allocated
10->1
11->1
12->1
```

VIVA QUESTIONS

1) List the various types of files
2) What are the various file allocation strategies?
3) What is linked allocation?
4) What are the advantages of linked allocation?
5) What are the disadvantages of sequential allocation methods?

Lab Program 10
Develop a C program to simulate SCAN disk scheduling algorithm.

DESCRIPTION
One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling.
SCAN algorithm: The disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
        clrscr();
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<n+2;i++)
        {
        for(j=0;j<(n+2)-i-1;j++)
                {
                        if(t[j]>t[j+1])
                        {
                        temp=t[j];
                        t[j]=t[j+1];
                        t[j+1]=temp;
                        }
                }
        }
        for(i=0;i<n+2;i++)
        if(t[i]==h)
                j=i;
                k=i;
                p=0;
        while(t[j]!=0)
        {
        atr[p]=t[j]; j--;
        p++;
        }
        atr[p]=t[j];
        for(p=k+1;p<n+2;p++,k++)
        atr[p]=t[k+1];
        for(j=0;j<n+1;j++)
```

```
            {
                        if(atr[j]>atr[j+1])
                        d[j]=atr[j]-atr[j+1];
                        else
                        d[j]=atr[j+1]-atr[j];
                        sum+=d[j];
            }
            printf("\nAverage header movements:%f",(float)sum/n);
            getch();
}
```

INPUT

```
Enter no. of tracks: 9
Enter track position: 55 58 60 70 18 90 150 160 184
```

OUTPUT

```
Tracks traversed            Difference between tracks
150                         50
160                         10
184                         24
90                          94
70                          20
60                          10
58                           2
55                          3
18                          37

Average header movements: 27.77
```

1)First Come First Serve

Basic Theory :

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival.

It is non-preemptive algorithm. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue. FCFS also suffers from starvation which arise if the first process has larger burst time.

STEP 1 :

Input the number of processes required to be scheduled using FCFS, burst time for each process and its arrival time.

STEP 2 :

Using enhanced bubble sort technique, sort the all given processes in ascending order according to arrival time in a ready queue.

STEP 3 :

Calculate the Finish Time, Turn Around Time and Waiting Time for each process which in turn help to calculate Average Waiting Time and Average Turn Around Time required by CPU to schedule given set of process using FCFS.

- process using FCFS.

PLAY

RESET

Average TAT :0

Average WT :0

Average RT :0

Arrival Time :

Burst Time :

ADD

PID	AT	BT	ST	CT	RT	WT	TAT

PLAY

RESET

Average TAT :2.00

Average WT :0.00

Average RT :0.00

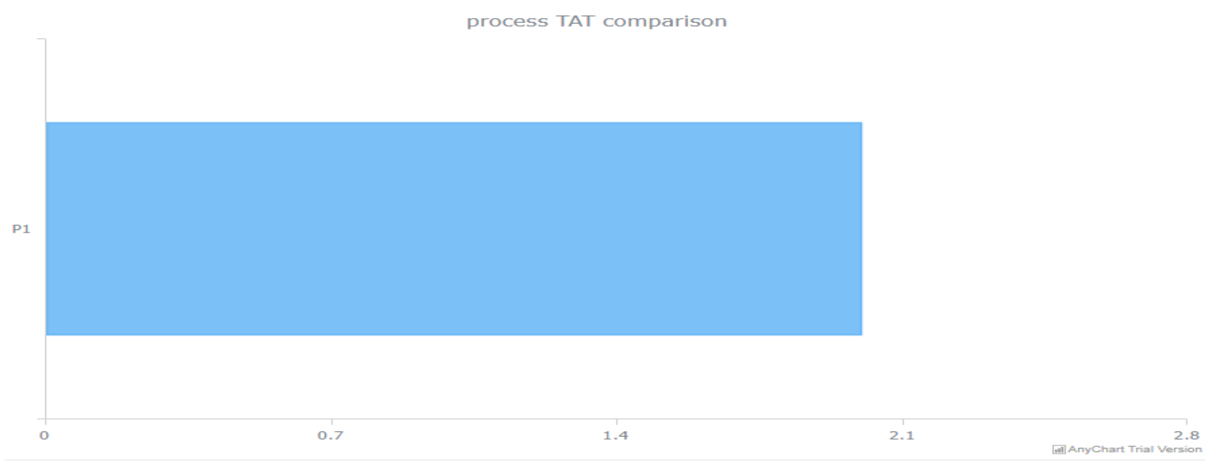
Arrival Time : 3

Burst Time : 5

ADD

PID	AT	BT	ST	CT	RT	WT	TAT	
P1	3	2	3	5	0	0	2	DEL

Start Time	P1
3	5



2)Round-robin scheduling,

Basic Theory :

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

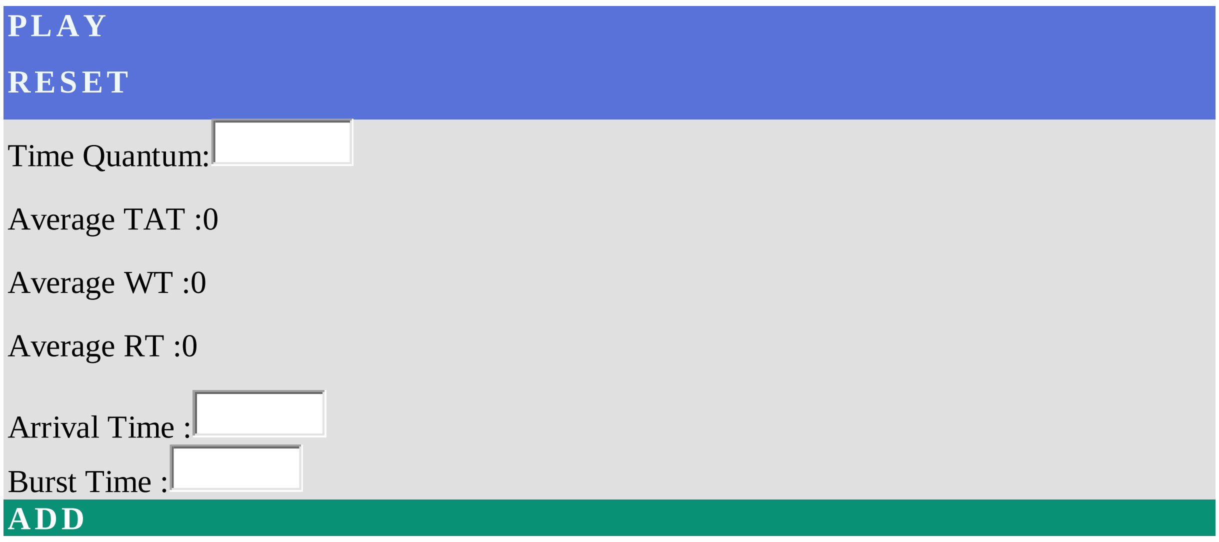
The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking. One of the most commonly used technique in CPU scheduling as a core. It is preemptive as processes are assigned CPU only for a fixed slice of time at most.

STEP 1 :
Assume that initially there are no ready processes, when the first one, A, arrives. It has priority 0 to begin with. Since there are no other accepted processes, A is accepted immediately.

STEP 2 :
After a while another process, B, arrives. As long as b / a < 1, B's priority will eventually catch up to A's, so it is accepted; now both A and B have the same priority.

STEP 3 :
All accepted processes share a common priority (which rises at rate b); that makes this policy easy to implement i.e any new process's priority is bound to get accepted at some point. So no process has to experience starvation.

STEP 4 :



Time Quantum:

Average TAT :0

Average WT :0

Average RT :0

Arrival Time :

Burst Time :

- Even if b / a > 1, A will eventually finish, and then B can be accepted.

PLAY

RESET

Time Quantum:

Average TAT :17.50

Average WT :4.00

Average RT :2.50

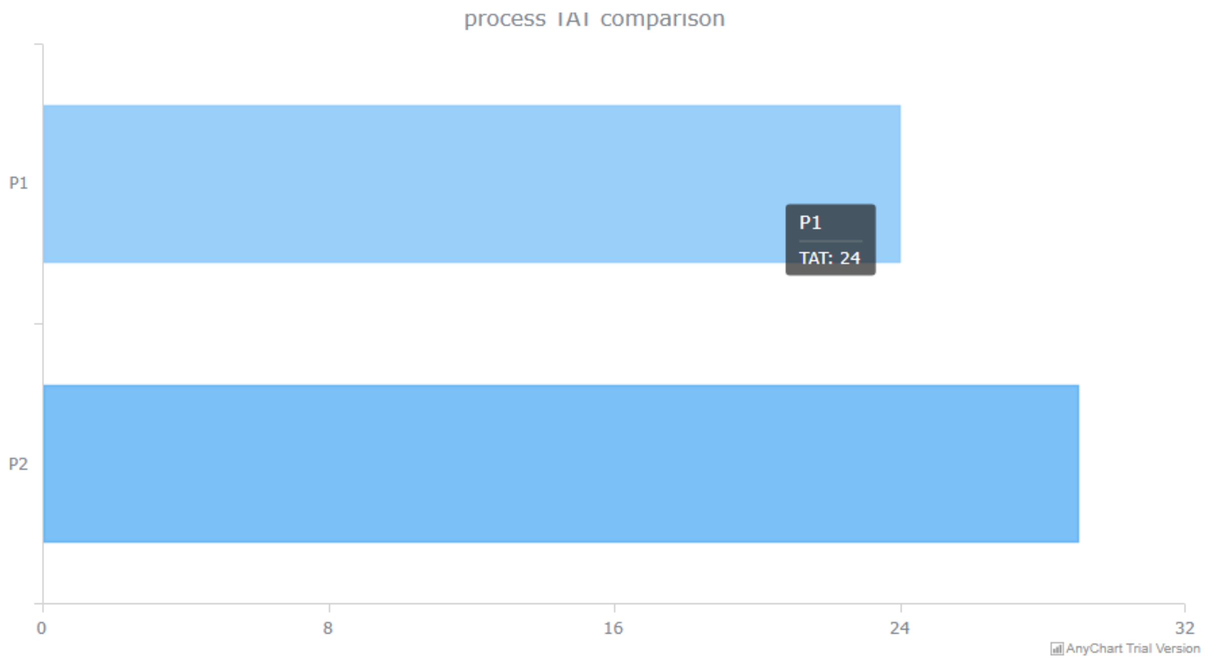
Arrival Time :

Burst Time :

ADD

PID	AT	BT	ST	CT	RT	WT	TAT	
P1	0	24	0	27	0	3	27	DEL
P2	0	3	5	8	5	5	8	DEL

Start Time	P1	P2	P1	P1	P1	P1
0	5	8	13	18	23	27


process TAT comparison
P1
P2
P1
TAT: 24
AnyChart Trial Version