EXPERIMENT NO.1

AIM: Demonstrate the process creation and termination using system calls -fork(), vfork(), exit(), return0

DESCRIPTION:

fork():
The fork() system call is used to create a new process by duplicating the existing process. It creates a copy of the calling process, known as the child process, which is almost identical to the parent process. The child process gets a unique process identifier (PID) and runs independently of the parent process. The child process inherits various attributes of the parent, such as open file descriptors, memory space, and environment variables. The fork() system call returns different values in the parent and child processes. In the parent process, it returns the PID of the child process, while in the child process, it returns 0.

vfork():
The vfork() system call is similar to fork() but has some differences. When vfork() is called, it creates a child process that shares the parent's memory space rather than duplicating it. This means that the child process executes in the same address space as the parent process. The purpose of vfork() is primarily to create a child process for quick execution of a new program using the exec() system call. In vfork(), the child process executes before the parent process resumes execution. However, the child process must be cautious about modifying any shared memory, as it may lead to unexpected results. Like fork(), vfork() returns the PID of the child process in the parent process and 0 in the child process.

exit():
The exit() system call is used to terminate the calling process. It is called at the end of a program or when a specific condition is met that requires immediate termination. When exit() is invoked, it performs various cleanup tasks, such as closing open file descriptors and freeing allocated memory. It also returns an exit status to the parent process, which can be used to determine the outcome of the terminated process. By convention, an exit status of 0 indicates successful termination, while non-zero values indicate different types of errors or abnormal terminations.

return 0:
return 0 is not a system call but rather a statement commonly used in programming languages like C and C++. It is used to indicate the successful execution of a program or function. When a program or function reaches the return 0 statement, it signifies that it has completed its execution without encountering any errors or exceptions. It is a way to communicate to the calling process that the program or function executed successfully. The value 0 can be used as an exit status to indicate success, following the convention mentioned earlier.

ALGORITHM

STEP1: Define the main function.
STEP2: Declare variables pid and vpid of type pid_t to store the process IDs, and status of type int to store termination status.
STEP3: Call the fork() system call to create a child process. Assign the return value of fork() to the pid variable.
STEP4: Check the value of pid using an if-else statement:

If pid is 0, it means that the current process is the child process. Proceed with the child process block.
If pid is a positive value, it represents the child process ID in the parent process. Proceed with the

parent process block.
If pid is -1, it indicates that forking failed. In this case, print an error message using fprintf() and return 1 to indicate an error.

STEP5: Inside the child process block (fork):
Print a message indicating that it is the child process created using fork() and display its process ID using getpid().
Print a message indicating that the child process is exiting.
Terminate the child process using exit(0).

STEP6: Inside the parent process block (fork):
Print a message indicating that it is the parent process and display the child process ID obtained from the pid variable.
Print a message indicating that the parent process is waiting for the child to terminate.
Use wait(&status) to wait for the child process to terminate. Store the termination status in the status variable.
Print a message indicating that the child process has terminated.

STEP7: Call the vfork() system call to create a child process. Assign the return value of vfork() to the vpid variable.

STEP8: Check the value of vpid using an if-else statement:
If vpid is 0, it means that the current process is the child process. Proceed with the child process block.
If vpid is a positive value, it represents the child process ID in the parent process. Proceed with the parent process block.
If vpid is -1, it indicates that vfork() failed. In this case, print an error

STEP9: Inside the child process block (vfork):
Print a message indicating that it is the child process created using vfork() and display its process ID using getpid().
Print a message indicating that the child process is exiting.
Terminate the child process using _exit(0).

STEP10: Inside the parent process block (vfork):
Print a message indicating that it is the parent process and display the child process ID obtained from the vpid variable.
Print a message indicating that the parent process is waiting for the child to terminate.
Use wait(&status) to wait for the child process to terminate. Store the termination status in the status variable.
Print a message indicating that the child process has terminated.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid, vpid;
    int status;

    // Fork system call
```

```
    pid = fork();

    if (pid == 0) {
        // Child process
        printf("Child process created using fork. Process ID: %d\n",
getpid());
        printf("Child process exiting.\n");
        exit(0);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process. Child process ID: %d\n", pid);
        printf("Parent process waiting for child to terminate.\n");
        wait(&status);
        printf("Child process terminated.\n");
    } else {
        // Fork failed
        fprintf(stderr, "Fork failed.\n");
        return 1;
    }

    // Vfork system call
    vpid = vfork();

    if (vpid == 0) {
        // Child process
        printf("Child process created using vfork. Process ID: %d\n",
getpid());
        printf("Child process exiting.\n");
        _exit(0);
    } else if (vpid > 0) {
        // Parent process
        printf("Parent process. Child process ID: %d\n", vpid);
        printf("Parent process waiting for child to terminate.\n");
        wait(&status);
        printf("Child process terminated.\n");
    } else {
        // Vfork failed
        fprintf(stderr, "Vfork failed.\n");
        return 1;
    }

    return 0;
}
```

OUTPUT:

```
Child process created using fork. Process ID: 1234
Child process exiting.
Parent process. Child process ID: 1234
Parent process waiting for child to terminate.
Child process terminated.
Child process created using vfork. Process ID: 1235
Child process exiting.
Parent process. Child process ID: 1235
Parent process waiting for child to terminate.
Child process terminated.
```

EXPERIMENT NO.2

AIM: Simulate the following CPU Scheduling Algorithms –

1. FCFS
2. SJF
3. Priority
4. Round Robin

FIRST COME FIRST SERVE:
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 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 processes
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 27.000000
Time --
```

SHORTEST JOB FIRST:
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 Burt time
Step 7: For each process in the ready queue, Calculate

- Waiting time(n)= waiting time (n-1) + Burst time (n-1)
- Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

- Average waiting time = Total waiting Time / Number of process
- 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                    4
processes --
Enter Burst Time for                   6
Process 0 --
Enter Burst Time for                   8
Process 1 --
Enter Burst Time for                   7
Process 2 --
Enter Burst Time for                   3
Process 3 --
```

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNARO UND 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	

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 assigns 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
```

PRIORITY:

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

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITINGTIME	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

EXPERIMENT NO.3

AIM: Demonstrate the concepts of Producer-Consumer, Reader Writer, Dining Philosophers Problems using Semaphores.

DESCRIPTION

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

SOURCE CODE

PRODUCER-CONSUMER

```
#include<stdio.h>
void main()
{
int buffer[10], bufsize, in, out, produce, consume,
choice=0; in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice: ");
scanf("%d",&choice);
switch(choice) {
case 1: if((in+1)%bufsize==out)
printf("\nBuffer is Full");
else
{
}
break;;
printf("\nEnter the value: ");
```

```
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
case 2: if(in == out)
printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
break;
} } }
```

OUTPUT

```
1. Produce     2. Consume     3.   Exit
Enter your choice: 2
Buffer is Empty
1. Produce     2. Consume     3.   Exit
Enter your choice: 1
Enter the value: 100
1. Produce     2. Consume     3.   Exit
Enter your choice: 2
The consumed value is 100
1. Produce     2. Consume     3.   Exit
Enter your choice: 3
```

Dining Philosophers

Description

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

SOURCE CODE

```
#include<stdio.h>
#include<conio.h>
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
int i; clrscr();
printf("\n\nDINING PHILOSOPHER PROBLEM");
printf("\nEnter the total no. of philosophers: ");
scanf("%d",&tph);
for(i=0;i<tph;i++)
{
philname[i]=(i+1); status[i]=1;
}
printf("How many are hungry : ");
scanf("%d", &howhung);
if(howhung==tph)
{
printf("\n All are hungry..\nDead lock stage will occur");
printf(\n"Exiting\n");
else{
for(i=0;i<howhung;i++){
printf("Enterphilosopher%dposition:",(i+1));
scanf("%d",&hu[i]);
status[hu[i]]=2;
}
do
{
printf("1.One can eat at a time\t2.Two can eat at a time
\t3.Exit\nEnter your choice:");
scanf("%d", &cho);
switch(cho)
{
case 1: one();
break;
case 2: two();
break;
case 3: exit(0);
default: printf("\nInvalid option..");
}
}while(1);
}
}
```

```
one()
{
int pos=0, x, i;
printf("\nAllow one philosopher to eat at any time\n");
for(i=0;i<howhung; i++, pos++)
{
printf("\nP %d is granted to eat", philname[hu[pos]]);
for(x=pos;x<howhung;x++)
printf("\nP %d is waiting", philname[hu[x]]);
}
}
two()
{
int i, j, s=0, t, r, x;
printf("\n Allow two philosophers to eat at same
time\n"); for(i=0;i<howhung;i++)
{
for(j=i+1;j<howhung;j++)
{
if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
{
printf("\n\ncombination %d \n", (s+1));
t=hu[i];
r=hu[j]; s++;
printf("\nP %d and P %d are granted to eat", philname[hu[i]],
philname[hu[j]]);
for(x=0;x<howhung;x++)
{
if((hu[x]!=t)&&(hu[x]!=r))
printf("\nP %d is waiting", philname[hu[x]]);
}
}
}
}
}
```

INPUT

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5
```

OUTPUT

```
1. One can eat at a time 2.Two can eat at a time  3.Exit  Enter
your choice: 1

Allow one philosopher to eat at any time P 3 is granted to eat
P 3 is waiting P 5 is waiting P 0 is waiting
P 5 is granted to eat P 5 is waiting
P 0 is waiting
P 0 is granted to eat P 0 is waiting
1.One can eat at a time  2.Two can eat at a time 3. Exit Enter
your choice: 2

Allow two philosophers to eat at same time combination 1
P 3 and P 5 are granted to eat P 0 is waiting

combination 2
P 3 and P 0 are granted to eat P 5 is waiting

combination 3
P 5 and P 0 are granted to eat P 3 is waiting

1.One can eat at a time  2.Two can eat at a time  3.Exit  Enter
your choice: 3
```

Reader Writer

Description

A reader writer is a term commonly used in the context of concurrent programming and computer science. It refers to a synchronization problem that arises when multiple processes or threads are accessing a shared resource, such as a file or a data structure, where some processes only read from the resource while others may also write to it.

SOURCE CODE

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
sem_t mutex,writeblock;
int data = 0,rcount = 0;

void *reader(void *arg)
{
int f;
f = ((int)arg);
sem_wait(&mutex);
```

```
rcount = rcount + 1;
if(rcount==1)
sem_wait(&writeblock);
sem_post(&mutex);
printf("Data read by the reader%d is %d\n",f,data);
sleep(1);
sem_wait(&mutex);
rcount = rcount - 1;
if(rcount==0)
sem_post(&writeblock);
sem_post(&mutex);
}

void *writer(void *arg)
{
int f;
f = ((int) arg);
sem_wait(&writeblock);
data++;
printf("Data writen by the writer%d is %d\n",f,data);
sleep(1);
sem_post(&writeblock);
}

main()
{
int i,b;
pthread_t rtid[5],wtid[5];
sem_init(&mutex,0,1);
sem_init(&writeblock,0,1);
for(i=0;i<=2;i++)
{
pthread_create(&wtid[i],NULL,writer,(void *)i);
pthread_create(&rtid[i],NULL,reader,(void *)i);
}
for(i=0;i<=2;i++)
{
pthread_join(wtid[i],NULL);
pthread_join(rtid[i],NULL);
}
}
```

OUTPUT

```
root@bt:~# gcc rwp2.c -lpthread
rwp2.c: In function 'reader':
rwp2.c:11: warning: cast from pointer to integer of different size
rwp2.c: In function 'writer':
rwp2.c:29: warning: cast from pointer to integer of different size
rwp2.c: In function 'main':
rwp2.c:45: warning: cast to pointer from integer of different size
rwp2.c:46: warning: cast to pointer from integer of different size
root@bt:~# ./a.out
Data read by the reader2 is 0
Data read by the reader1 is 0
Data read by the reader0 is 0
Data writen by the writer2 is 1
Data writen by the writer1 is 2
Data writen by the writer0 is 3
root@bt:~#
```

EXPERIMENT: 4
AIM: Apply Bankers Algorithm for the purpose of deadlock avoidance.

DESCRIPTION:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.
Why Banker's algorithm is named so?
Banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.
The following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resource types.
Available :

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

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process **Pi** may request at most **'k'** instances of resource type **Rj.**

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process **Pi** is currently allocated **'k'** instances of resource type **Rj**

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process **Pi** currently needs **'k'** instances of resource type **Rj**
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.
Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.
Safety Algorithm
The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n
2) Find an i such that both
a) Finish[i] = false
b) Needi <= Work
if no such i exists goto step (4)
3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)
4) if Finish [i] = true for all i
then the system is in a safe state

Resource-Request Algorithm
Let Requesti be the request array for process Pi. Requesti [j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1) If Requesti <= Needi Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2) If Requesti <= Available Goto step (3); otherwise, Pi must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Requesti
Allocationi = Allocationi + Requesti
Needi = Needi– Requesti

Example:
Considering a system with five processes P0 through P4 and three resources of type A,

B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

SOURCE CODE:

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

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

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

    int avail[3] = { 3, 3, 2 }; // Available Resources
 int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
```

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

                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                        f[i] = 1;
                }
            }
        }
    }

      int flag = 1;

      for(int i=0;i<n;i++)
    {
      if(f[i]==0)
      {
        flag=0;
         printf("The following system is not safe");
        break;
      }
    }

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

EXPERIMENT NO:5
AIM: Demonstrate different page replacement algorithms – FIFO, LRU, and OPTIMAL
FIFO
DESCRIPTION

The operating system uses the method of paging for memory management. This method involves page replacement algorithms to make a decision about which pages should be replaced when new pages are demanded. The demand occurs when the operating system needs a page for processing, and it is not present in the main memory. The situation is known as a page fault

In this situation, the operating system replaces an existing page from the main memory by bringing a new page from the secondary memory.

In such situations, the FIFO method is used, which is also refers to the First in First Out concept. This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

ALGORITHM

1. Start the process

2. Declare the size with respect to page length

3. Check the need of replacement from the page to memory

4. Check the need of replacement from old page to new page in memory

5. Forma queue to hold all pages

6. Insert the page require memory into the queue

7. Check for bad replacement and page fault

8. Get the number of processes to be inserted

9. Display the values

10. Stop the process

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
```

LEAST RECENTLY USED
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
```

OPTIMAL
ALGORTHIM:

1. Start Program
2. Read Number of Pages and Frames
3.Read Each Page Value
4. Search for Page in The Frames
5.If Not Available Allocate Free Frame
6. If No Frames Is Free Replaced the Page with The Page That Is Leastly Used
7.Print Page Number of Page Faults
8.Stop process.

SOURCE CODE:

```
/* Program to simulate optimal page replacement */
#include<stdio.h>
#include<conio.h>
int fr[3], n, m;
void
display();
void main()
{
int i,j,page[20],fs[10];
int
max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0;
float pr;
clrscr();
printf("Enter length of the reference string: ");
scanf("%d",&n);
printf("Enter the reference string: ");
for(i=0;i<n;i++)
scanf("%d",&page[i]);
```

```
printf("Enter no of frames: ");
scanf("%d",&m);
for(i=0;i<m;i++)
fr[i]=-1; pf=m;
for(j=0;j<n;j++) {
flag1=0; flag2=0;
for(i=0;i<m;i++) {
if(fr[i]==page[j]) {
flag1=1; flag2=1;
break; }}
if(flag1==0) {
for(i=0;i<m;i++) {
if(fr[i]==
-1)
{
fr[i]=page[j]; flag2=1;
break; }}}
if(flag2==0) {
for(i=0;i<m;i++)
lg[i]=0;
for(i=0;i<m;i++) {
for(k=j+1;k<=n;k++) {
if(fr[i]==page[k]) {
lg[i]=k
-j;
break; }}}
found=0;
for(i=0;i<m;i++) {
if(lg[i]==0) {
index=i;
found = 1;
break;
}
}
if(found==0)
{
max=lg[0]; index=0;
for(i=0;i<m;i++)
{
if(max<lg[i])
{
max=lg[i];
index=i;
}
}
}
fr[index]=page[j];
```

```
pf++;
}
display();
}
printf("Number of page faults : %d\n", pf);
pr=(float)pf/n*100;
printf("Page fault rate = %f \n", pr); getch();
}
void display()
{
int i; for(i=0;i<m;i++)
printf("%d\t",fr[i]);
printf("\n");
}
```

OUTPUT:

```
Enter length of the reference string: 12
Enter the reference string: 1 2 3 4 1 2 5 1 2 3 4 5
Enter no of frames: 3
1 -1 -1
1 2 -1
1 2 3
1 2 4
1 2 4
1 2 4
1 2 5
1 2 5
1 2 5
3 2 5
4 2 5
4 2 5
Number of page faults : 7 Page fault rate = 58.333332
```

EXPERIMENT NO:6
AIM: Compare different disk scheduling algorithms-FCFS,SCAN,C-SCAN.

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. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the 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. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

SOURCE CODE:

A) FCFS DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
int t[20], n, I, j, tohm[20], tot=0; float avhm;
clrscr();
printf("enter the no.of tracks");
scanf("%d",&n);
```

```
printf("enter the tracks to be traversed");
for(i=2;i<n+2;i++)
scanf("%d",&t*i+);
for(i=1;i<n+1;i++)
{
tohm[i]=t[i+1]-t[i];
if(tohm[i]<0)
tohm[i]=tohm[i]*(-1);
}
for(i=1;i<n+1;i++)
tot+=tohm[i];
avhm=(float)tot/n;
printf("Tracks traversed\tDifference between tracks\n");
for(i=1;i<n+1;i++)
printf("%d\t\t\t%d\n",t*i+,tohm*i+);
printf("\nAverage header movements:%f",avhm);
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
55	45
58	3
60	2
70	10
18	52
90	72
150	60
160	10
184	24

```
Average header movements:30.888889
```

B) SCAN DISK SCHEDULING ALGORITHM

```
#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

C) C-SCAN DISK SCHEDULING ALGORITHM

```
#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 total tracks");
scanf("%d",&tot);
t[2]=tot-1;
printf("enter the tracks");
for(i=3;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;break;
p=0;
while(t[j]!=tot-1)
{
atr[p]=t[j];
j++;
p++;
}
atr[p]=t[j];
p++;
i=0;
while(p!=(n+3) && t[i]!=t[h])
{
atr[p]=t[i]; i++;
p++;
}
for(j=0;j<n+2;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("total header movements%d",sum);
printf("avg is %f",(float)sum/n);
getch();
}
```

INPUT

Enter the track position : 55 58 60 70 18 90 150 160 184
Enter starting position : 100

OUTPUT

Tracks traversed	Difference Between tracks
150	50
160	10
184	24
18	240
55	37
58	3
60	2
70	10
90	20

Average seek time : 35.7777779

EXPERIMENT NO.7

AIM: Compare and simulate file allocation strategies—indexed and sequential approaches.

Description:

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

Sequential:

Algorithm:

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 to each in sequential order a).

Randomly select a location from availablelocation s1= random(100);

Check whether the required locations are free from the selected location.

if(b[s1].flag==0){

for (j=s1;j<s1+p[i];j++){
if((b[j].flag)==0)count++;
}
if(count==p[i]) break;
}
b) Allocate and set flag=1 to the allocated locations. for(s=s1;s<(s1+p[i]);s++)
{
k[i][j]=s; j=j+1; b[s].bno=s;
b[s].flag=1;
}
Step 5: Print the results file no, length, Blocks allocated. Step
6: Stop the program

SOURCE CODE :

```
#include<stdio.h>
main()
{
int f[50],i,st,j,len,c,k;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
X:
printf("\n Enter the starting block & length of file");
scanf("%d%d",&st,&len);
for(j=st;j<(st+len);j++)
if(f[j]==0)
{
f[j]=1
;
printf("\n%d->%d",j,f[j]);
}
else
{
printf("Block already allocated");
break;
}
if(j==(st+len))
printf("\n the file is allocated to disk");
printf("\n if u want to enter more files?(y-1/n-0)");
```

```
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
getch();
}
```

OUTPUT:

```
Enter the starting block & length of file 4 10
4->1
5->1
6->1
7->1
8->1
9->1
10->1
11->1
12->1
13->1
The file is allocated to disk.
```

INDEXED:

DESCRIPTION:

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

ALGORITHM:

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.

```
q=random(100);
{
if(b[q].flag==0)
b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
```

Step 5: Print the results file no, length ,Blocks allocated.
Step 6: Stop the program

SOURCE CODE :

```
#include<stdio.h>
int f[50],i,k,j,inde[50],n,c,count=0,p;
main() {
clrscr();
for(i=0;i<50;i++)
f[i]=0;
x: printf("enter index block
\t");

scanf("%d",&p);
if(f[p]==0)
{
f[p]=1;
printf("enter no of files on index
\t");

scanf("%d",&n); }
else {
printf("Block already allocated
\n");

goto x; }
for(i=0;i<n;i++)
scanf("%d",&inde[i]);
for(i=0;i<n;i++)
if(f[inde[i]]==1) {
printf("Block already allocated");
goto x; }
for(j=0;j<n;j++)
f[inde[j]]=1;
printf("
\n allocated");
printf("
\n file indexed");
for(k=0;k<n;k++)
printf("
\n %d
->%d:%d",p,inde[k],f[inde[k]]);
printf(" Enter 1 to enter more files and 0 to exit
\t");

scanf("%d",&c);
if(c==1)
goto x;
else
exit();
getch();
}
```

OUTPUT:

```
enter index block 9
Enter no of files on index 3 1
2 3
Allocated
File indexed
9->1:1
9->2;1
9->3:1 enter 1 to enter more files and 0 to exit
```

EXPERIMENT NO.8

AIM: Demonstrate the OS installation, OS protection and security—Firewall, protections, threat detections and prevention.

Description:

OS Installation:

- Obtain the installation media: This can be a DVD, USB drive, or downloaded ISO file.
- Boot from the installation media: Configure your computer's BIOS to boot from the installation media and follow the on-screen instructions to start the installation process.
- Select the installation type: Choose whether to perform a clean installation or upgrade an existing OS.
- Partitioning: Configure disk partitions and allocate space for the operating system.
- Follow the installation prompts: This includes selecting language preferences, agreeing to the license agreement, and specifying additional settings.
- Complete the installation: Once the installation process finishes, restart your computer and boot into the newly installed operating system.

OS Protection:

- Keep the OS up to date: Regularly install security patches and updates provided by the OS vendor to protect against known vulnerabilities.
- Use strong passwords: Set strong and unique passwords for user accounts to prevent unauthorized access.
- User account management: Limit user privileges and create separate accounts for different users to minimize potential risks.
- Enable a firewall: Activate the built-in firewall or install a third-party firewall software to monitor and control incoming and outgoing network traffic.
- Install antivirus software: Deploy reputable antivirus software and keep it updated to scan for and remove malware.
- Data backup: Regularly back up your important data to ensure you can recover in the event of system failure or data loss.

Firewall:

- Operating System Firewall: Most modern operating systems come with a built-in firewall. Enable the firewall and configure the desired rules to allow or block specific network connections.
- Third-party Firewalls: You can also install third-party firewall software that provides additional features and customization options.
- Threat Detection and Prevention:
- Antivirus and Antimalware: Install reputable antivirus and antimalware software to scan and detect malicious software on your system. Keep the software updated to ensure it can recognize the latest threats.
- Intrusion Detection Systems (IDS): Implement IDS software or hardware that monitors network traffic for suspicious activities and alerts you in case of potential threats.
- Security Updates: Regularly update your operating system, applications, and plugins to patch known vulnerabilities and protect against emerging threats.
- User Education: Educate yourself and your users about safe computing practices, such as avoiding suspicious downloads, not clicking on unknown links, and being cautious with email attachments.