PROGRAM 1

Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

System calls allow programs to create and terminate processes, as well as manage inter-process communication. Security: System calls provide a way for programs to access privileged resources, such as the ability to modify system settings or perform operations that require administrative permissions.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include
<sys/wait.h>
int main()
{
// Variable to store the process ID.
pid_t child_pid; char ch[3];
// Create a child process using
fork().child_pid = fork();
if (child_pid == -1) {
// Error occurred during fork.
perror("fork");
exit(EXIT_FAILURE);
}
if (child_pid == 0)
{
// This code executes in the child process.
printf("Child process (PID: %d) is running.\n", getpid());
// Execute a different program using exec().
//abort(); child terminated abnormally by sending signal number 6 to the parent
// return(-1); child terminates with exit status 255
execl("/bin/date","date", NULL);
exit(0);
}
else {
// This code executes in the parent process.
printf("Parent process (PID: %d) is waiting for the child to terminate.\n", getpid());
// Wait for the child process to terminate using wait().
int status;
wait(&status);
printf("parent resumes\n");
if (WIFEXITED(status))
{
printf("\nChild process (PID: %d) terminated with status %d.\n", child_pid, WEXITSTATUS(status));
}
else if (WIFSIGNALED(status))
{
printf("\nChild process (PID: %d) terminated due to signal %d.\n", child_pid, WTERMSIG(status));
}
else
{
```

```
printf("\nChild process (PID: %d) terminated abnormally.\n", child_pid);
}
}
return 0;
}
```

Output:

```
Parent process (PID: 11284) is waiting for the child to terminate.
Child process (PID: 11285) is running.
parent resumes

Child process (PID: 11285) terminated with status 255.
Parent process (PID: 11356) is waiting for the child to terminate.
Child process (PID: 11357) is running.
parent resumes

Child process (PID: 11357) terminated due to signal 6.
Parent process (PID: 11433) is waiting for the child to
terminate.Child process (PID: 11434) is running.
Tue Oct 10 10:50:07 IST 2023
parent resumes

Child process (PID: 11434) terminated with status 0.
```

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) FCFS

FCFS Scheduling Algorithm

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.

```
#include<stdio.h>
int main()
{
int bt[20], wt[20], tat[20],i, n;float wtavg, tatavg;
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);
}
```

Output:

```
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
```

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) SJF

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. The Shortest Job First Scheduling Algorithm chooses the process that has the smallest next CPU burst.

```
#include<stdio.h>
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
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);
}
```

Output:

```
Enter the number of processes: 3
Enter Burst Time for Process 0: 10
Enter Burst Time for Process 1: 8
Enter Burst Time for Process 2: 7
```

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P2	7	0	7
P1	8	7	15
P0	10	15	25

```
Average Waiting Time -- 7.333333
Average Turnaround Time -- 15.666667
```

c) Round Robin

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. This scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes.

```
#include<stdio.h>
int main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
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]);
}
```

Output:

```
Enter the no of processes -- 3
Enter Burst Time for process 1 -- 10
Enter Burst Time for process 2 -- 8
Enter Burst Time for process 3 -- 7
Enter the size of time slice -- 5
```

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	10	10	20
2	8	15	23
3	7	18	25

```
The Average Turnaround time is -- 22.666666
The Average Waiting time is -- 14.333333
```

Enter the no of processes -- 3
Enter Burst Time for process 1 -- 10
Enter Burst Time for process 2 -- 8
Enter Burst Time for process 3 -- 7
Enter the size of time slice – 4

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	10	15	25
2	8	12	20
3	7	16	23

The Average Turnaround time is -- 22.666666

The Average Waiting time is -- 14.333333

d) Priority

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.

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

Output:

```
Enter the number of processes --- 3
Enter the Burst Time & Priority of Process 0 --- 10
3Enter the Burst Time & Priority of Process 1 --- 8
1 Enter the Burst Time & Priority of Process 2 --- 7
2
```

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	8	0	8
2	2	7	8	15
0	3	10	15	25

```
Average Waiting Time is --- 7.666667
Average Turnaround Time is --- 16.000000
```

PROGRAM 3

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

//Producer Consumer

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>
int mutex = 1;
int full = 0;
int empty = 3, x = 0;
void producer()
{
--mutex;
++full;
--
empty;
x++;
printf("\nProducer produces item %d",x);
++mutex;
}
void consumer()
{
--mutex;
--full;
++empty;
printf("\nConsumer consumes item %d",x);
x--;
++mutex;
}
int main()
{
int n, i;
printf("\n1. Press 1 for Producer""\n2. Press 2 for Consumer" "\n3. Press 3 for Exit");
```

#pragma omp critical

```
for (i=1;i>0;i++)
{
   printf("\nEnter your choice:");
   scanf("%d", &n);
   // Switch Cases
   switch (n)
```

```
{
case 1:
if ((mutex == 1) && (empty != 0))
{
producer();
}
else
{
printf("Buffer is full!");
}
break;
case 2:
if ((mutex == 1)&& (full != 0))
{
consumer();
}
else
{
printf("Buffer is empty!");
}
break;
case 3:
exit(0);
break;
}
}
}
```

Output:

```
Press 1 for Producer
Press 2 for Consumer
Press 3 for Exit

Enter your choice: 1
Producer produces item 1

Enter your choice: 1
Producer produces item 2

Enter your choice: 1
Producer produces item 3

Enter your choice: 1
Buffer is full!

Enter your choice: 2
Consumer consumes item 3
```

Enter your choice: 2
Consumer consumes item 2

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 2
Buffer is empty!

Enter your choice: 3

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 APIs in your program.

Inter-Process Communication (IPC) plays a vital role in the world of operating systems, enabling different processes to communicate and cooperate with each other. IPC is crucial for multi-tasking, enabling processes to share data, synchronize their activities, and collaborate effectively.

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

// Create separate files for reader and writer processes
// Unless the writer process writes into the buffer, reader cannot read

Writer Process

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include  <unistd.h>
int main()
{
int fd;
char buf[1024]="Hello BIT";
char * myfifo = "/ise/Desktop/tmp";
mkfifo(myfifo, 0666);
printf("Run Reader process to read the FIFO File\n");
fd = open(myfifo, O_WRONLY);
write(fd,buf,sizeof(buf));
close(fd);
unlink(myfifo); /* remove the FIFO */
return 0;
}
```

Reader Process

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define MAX_BUF 1024
int main()
{
 int fd;
 char *myfifo = "/ise /Desktop/tmp";
 char buf[MAX_BUF];
```

```
fd = open(myfifo, O_RDONLY);
read(fd, buf, MAX_BUF);
printf("Reader process has read : %s\n", buf);
close(fd);
return 0;
}
```

Output:

Run Reader process to read the FIFO File
Reader process has read: Hello BIT

PROGRAM 5

Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

The Bankers Algorithm in OS is a deadlock avoidance algorithm used to avoid deadlocks and to ensure safe execution of processes. The algorithm maintains a matrix of maximum and allocated resources for each process and checks if the system is in a safe state before allowing a process to request additional resources.

```
#include<stdio.h>
struct file
{
int all[10];
int max[10];
int need[10];
int flag;
};
void main()
{
struct file f[10];
int fl;
int i, j, k, p, b, n, r, g, cnt=0, id, newr;
int avail[10],seq[10];
printf("Enter number of processes -- ");
scanf("%d",&n);
printf("Enter number of resources -- ");
scanf("%d",&r);
for(i=0;i<n;i++)
{
printf("Enter details for P%d",i);
printf("\nEnter allocation\t -- \t");
for(j=0;j<r;j++)
scanf("%d",&f[i].all[j]);
printf("Enter Max\t\t -- \t");
for(j=0;j<r;j++)
scanf("%d",&f[i].max[j]);
f[i].flag=0;
}
printf("\nEnter Available Resources\t -- \t");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
 f[i].need[j]=f[i].max[j]-
f[i].all[j];if(f[i].need[j]<0)
f[i].need[j]=0;
}
}
cnt=0;
```

```
fl=0;
while(cnt!=n)
{
 g=0;
for(j=0;j<n;j++)
{
if(f[j].flag==0)
{
 b=0;
for(p=0;p<r;p++)
{
if(avail[p]>=f[j].need[p])b=b+1;
else
b=b-1;
}
if(b==r)
{
printf("\nP%d is visited",j);
seq[fl++]=j;
f[j].flag=1;
for(k=0;k<r;k++)
avail[k]=avail[k]+f[j].all[k];
cnt=cnt+1;
printf("(");
for(k=0;k<r;k++)
printf("%3d",avail[k]);
printf(")");
g=1;
}
}
}
if(g==0)
{
printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
printf("\n SYSTEM IS IN UNSAFE STATE");
goto y;
}
}
printf("\nSYSTEM IS IN SAFE STATE");
printf("\nThe Safe Sequence is -- (");
for(i=0;i<fl;i++)
printf("P%d ",seq[i]);
printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
for(i=0;i<n;i++)
{
printf("P%d\t",i);
for(j=0;j<r;j++)
printf("%6d",f[i].all[j]);
for(j=0;j<r;j++)
```

```
printf("%6d",f[i].max[j]);
for(j=0;j<r;j++)
printf("%6d",f[i].need[j]);
printf("\n");
}
}
```

Output:

```
Enter number of processes -- 5
Enter number of resources -- 3
Enter details for P0
        Enter allocation        --      0 1 0
        Enter Max               --      7 5 3
        Enter details for P1
        Enter allocation        --      2 0 0
        Enter Max               --      3 2 2
        Enter details for P2
        Enter allocation        --      3 0 2
        Enter Max               --      9 0 2
        Enter details for P3
        Enter allocation        --      2 1 1
        Enter Max               --      2 2 2
        Enter details for P4
        Enter allocation        --      0 0 2
        Enter Max               --      4 3 3

Enter Available Resources       --      3 3 2

        P1 is visited(          5       3       2)
        P3 is visited(          7       4       3)
        P4 is visited(          7       4       5)
        P0 is visited(          7       5       5)

P2 is visited( 10 5  7)
SYSTEM IS IN SAFE
STATE
The Safe Sequence is -- (P1 P3 P4 P0 P2 )
```

Process	Allocation			Max			Need		
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

PROGRAM 6

Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.

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.

a) Worst fit

```
#include<stdio.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];
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)
```

```
{
ff[i]=j; highest=temp;
}}}
frag[i]=highest;bf[ff[i]]=1; highest=0;
}
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]);
}
```

Output:

```
Memory Management Scheme - Worst Fit
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block1: 5
Block2: 2
Block3: 7
Enter the size of the
files:-
File1: 1
File2: 4
```

File_no	File_size	Block_no	Block_size	Fragement
1	1	3	7	6
2	4	1	5	1

b) Best fit

```
#include<stdio.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];
printf("\n\tMemory Management Scheme - Best 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)
if(lowest>temp)
{
ff[i]=j;
lowest=t
emp;
}}}
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();
}
```

Output:

```
Memory Management Scheme - Best Fit
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block1: 5
Block2: 2
Block3: 7
Enter the size of the
files :-
File1: 1
File2: 4
```

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

c) First Fit

```
#include<stdio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
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();
}
```

Output:

Memory Management Scheme - First Fit
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:-
File1: 1
File2: 4

File_no	File_size	Block_no	Block_size	Fragement
1	1	1	5	4
2	4	3	7	3

Memory Management Scheme - First Fit
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block1: 5
Block2: 2
Block3: 7
Enter the size of the files:-
File1: 4
File2: 1

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	4	1	5	1
2	1	2	2	1

PROGRAM 7

Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

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 OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

a) FIFO

```
#include<stdio.h>
void main()
{
int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
printf("\n Enter the length of reference string -- ");
scanf("%d",&n);
printf("\n Enter the reference string -- ");
for(i=0;i<n;i++)
scanf("%d",&rs[i]);
printf("\n Enter no. of frames -- ");
scanf("%d",&f);
for(i=0;i<f;i++)
m[i]=-1;
printf("\n The Page Replacement Process is -- \n");
for(i=0;i<n;i++)
{
for(k=0;k<f;k++)
{
if(m[k]==rs[i])
break;
}
if(k==f)
{
m[count++]=rs[i];
pf++;
}
for(j=0;j<f;j++)
printf("\t%d",m[j]);
if(k==f)
printf("\tPF No. %d",pf);
printf("\n");
if(count==f)
count=0;
}
printf("\n The number of Page Faults using FIFO are %d",pf);
}
```

Output:

```
Enter the length of reference string -- 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3
The Page Replacement Process is –

        7       -1      -1      PF No. 1
        7        0      -1      PF No. 2
        7        0       1      PF No. 3
        2        0       1      PF No. 4
        2        0       1
        2        3       1      PF No. 5
        2        3       0      PF No. 6
        4        3       0      PF No. 7
        4        2       0      PF No. 8
        4        2       3      PF No. 9
        0        2       3      PF No. 10
        0        2       3
        0        2       3
        0        1       3      PF No. 11
        0        1       2      PF No. 12
        0        1       2
        0        1       2
        7        1       2      PF No. 13
        7        0       2      PF No. 14
        7        0       1      PF No. 15

The number of Page Faults using FIFO are 15
```

b) LRU

```
#include<stdio.h>
int main()
{
int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
printf("Enter the length of reference string -- ");
scanf("%d",&n);
printf("Enter the reference string -- ");
for(i=0;i<n;i++)
{
scanf("%d",&rs[i]);
flag[i]=0;
}
printf("Enter the number of frames -- ");
scanf("%d",&f);
for(i=0;i<f;i++)
{
count[i]=0;
m[i]=-1;
```

```
}
printf("\nThe Page Replacement process is -- \n");
for(i=0;i<n;i++)
{
for(j=0;j<f;j++)
{
if(m[j]==rs[i])
{
flag[i]=1;
count[j]=next;
next++;
}
}
if(flag[i]==0)
{
if(i<f)
{
m[i]=rs[i];
count[i]=next;
next++;
}
else
{
min=0;
for(j=1;j<f;j++)
if(count[min] > count[j])
min=j;
m[min]=rs[i];
count[min]=next;
next++;
}
pf++;
}
for(j=0;j<f;j++)
printf("%d\t", m[j]);
if(flag[i]==0)
printf("PF No. -- %d", pf);
printf("\n");
}
printf("\nThe number of page faults using LRU are %d",pf);
}
```

Output:

Enter the length of reference string -- 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Enter the number of frames -- 3

The Page Replacement process is –

7	-1	-1	PF No. -- 1
7	0	-1	PF No. -- 2
7	0	1	PF No. -- 3
2	0	1	PF No. -- 4
2	0	1	
2	0	3	PF No. -- 5
2	0	3	
4	0	3	PF No. -- 6
4	0	2	PF No. -- 7
4	3	2	PF No. -- 8
0	3	2	PF No. -- 9
0	3	2	
0	3	2	
1	3	2	PF No. -- 10
1	3	2	
1	0	2	PF No. -- 11
1	0	2	
1	0	7	PF No. -- 12
1	0	7	
1	0	7	

The number of page faults using LRU are 12

PROGRAM 8

Simulate following File Organization Techniques
a) Single level directory b) Two level directory

a) Single level directory

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.

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;
void main()
{
int i,ch;
char f[30];
dir.fcnt = 0;
printf("\nEnter name of directory -- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n 1.Create File\t 2.Delete File\t 3.Search File \n 4.Display Files\t 5.Exit\n Enter 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 name of
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
file -- ");
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);
```

Output:

```
Enter name of directory -- dir1

1. Create File 2. Delete File 3. Search File 4. Display Files
Enter your choice: 1
Enter the name of the file: file1

1. Create File 2. Delete File 3. Search File 4. Display Files
Enter your choice: 1
Enter the name of the file: file2

1. Create File 2. Delete File 3. Search File 4. Display Files
Enter your choice: 4
The Files are -- file1    file2

1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice: 3
Enter the name of the file: file2
File file2 is 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: file2
File file2 is deleted

1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice: 3
Enter the name of the file: file2
File file2 is found
```

1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice: 4
The Files are -- file1
1. Create File 2. Delete File 3. Search File 4. Display Files 5. Exit
Enter your choice – 5

b) Two level directory

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

```
#include<stdio.h>
struct st
{
char dname[10];
char sdname[10][10];
char fname[10][10][10]; int ds,sds[10];
}
dir[10];
void main()
{
int i,j,k,n;
char name[10];
printf("enter number of users:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter user directory %d names:",i+1);
scanf("%s",&dir[i].dname);
printf("enter size of directories:");
scanf("%d",&dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("enter subdirectory name and size:");
scanf("%s",&dir[i].sdname[j]);
scanf("%d", &dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
{
printf("enter file name:");
scanf("%s",&dir[i].fname[j][k]);
printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");
printf("\n*******
for(i=0;i<n;i++)
{
{
printf("%s\t\t%d",dir[i].dname,dir[i].ds);
for(j=0;j<dir[i].ds;j++)
printf("\t%s\t\t%d\t", dir[i].sdname[j],dir[i].sds[j]);
```

```
for(k=0;k<dir[i].sds[j];k++)
printf("%s\t",dir[i].fname[j][k]);
printf("\n\t\t");
}
printf("\n");
}
```

Output:

```
Enter number of users: 2
Enter user directory1 names: user1
Enter size of directories: 2
Enter subdirectory name and size: dir1
2
Enter file name: f1
Enter file name: f2
Enter subdirectory name and size: dir2
3
Enter file name: f1
Enter file name: f2
Enter file name: f3
Enter user directory 2 names: user2
Enter size of directories: 3
Enter subdirectory name and size: dir1
2
Enter file name: f1
Enter file name: f3
Enter subdirectory name and size: dir2
2
Enter file name: f3
Enter file name: f4
Enter subdirectory name and size: dir3
4
Enter file name: fi
Enter file name: f2
Enter file name: f3
Enter file name: f4
```

dirname	size	subdirname	size	files			
user1	2	dir1	2	f1	f2		
		dir2	3	f1	f2	f3	
user2	3	dir1	2	f1	f3		
		dir2	2	f3	f4		
		dir3	4	fi	f2	f3	f4

PROGRAM 9

Develop a C program to simulate the Linked file allocation strategies.

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.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int f[50], p, i, st, len, j, c, k, a, fn=0;
for (i = 0; i < 50; i++)f[i] = 0;
/*printf("Enter how many blocks already allocated: ");
scanf("%d",&p);
printf("Enter blocks already allocated: ");
for (i=0;i<p; i++)
{
scanf("%d", &a);
f[a]=1;
}*/
x:
fn=fn+1;
printf("Enter index starting block and length: ");
scanf("%d%d", &st, &len);
k = len;
if (f[st]==0)
{
for(j=st;j<(st+k);j++)
{
if (f[j]==0)
{
f[j] = fn;
printf("%d ---------->%d\n", j, f[j]);
}
else
{
printf("%d Block is already allocated \n", j);
k++;
}}}
else
printf("%d starting block is already allocated \n", st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if (c == 1)
goto x;
else

exit(0);
return 0;
}
```

Output:

```
Enter index starting block and length: 1 3
1...>1
2...>1
3...>1
Do you want to enter more file(Yes - 1/No - 0)1
Enter index starting block and length: 5 3
5...>2
6...>2
7...>2
Do you want to enter more file(Yes - 1/No - 0)1
Enter index starting block and length: 4 5
4...>3
5 Block is already allocated
6 Block is already allocated
7 Block is already allocated
8...>3
9...>3
10...>3
11-------->3
Do you want to enter more file(Yes - 1/No - 0)0
```

PROGRAM 10

Develop a C program to simulate SCAN disk scheduling algorithm.

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.

// head movement towards left

```
#include<stdio.h>
int main()
{
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
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];
printf("seek sequence is:");
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("\n%d", atr[j]);
}
printf("\nTotal head movements:%d", sum);
}
```

Output:

```
Enter the no of tracks to be traversed 8
Enter the position of head 50
Enter the tracks 176
79
34
60
92
11
41
114
seek sequence is:
50
41
34
11
0
60
79
92
114
Total head movements: 226
```