LIST OF EXPERIMENTS

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

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

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

4 Develop a C program which demonstrates inter process communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

5 Develop a C program to simulate Bankers Algorithm for Dead Lock Avoidance.

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

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

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

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

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

SOFTWARE REQUIREMENTS & INSTALLATION PROCESS

Softwares:

1. Code blocks 10.05- IDE

Installation Process:

Step 1: Install the codeblocks 10.05(su –c yum install codeblocks.) or any higher version.

Step 2: Now open codeblocks

Step : Open a new project .

Experiment No: 1

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

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

Fork():

Theory: System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

- **Algorithm:**

- Step 1 − START.
- Step 2 – child process using system call-fork
- Step 3 − print hello.
- Step 4 − STOP

Program:

```
#include <stdio.h>

#include <sys/types.h>

int main()

{

fork();

fork();

fork();

printf("hello\n");

return 0;

}
```

Execution:

```
hello

hello

hello

hello

hello

hello

hello

hello
```

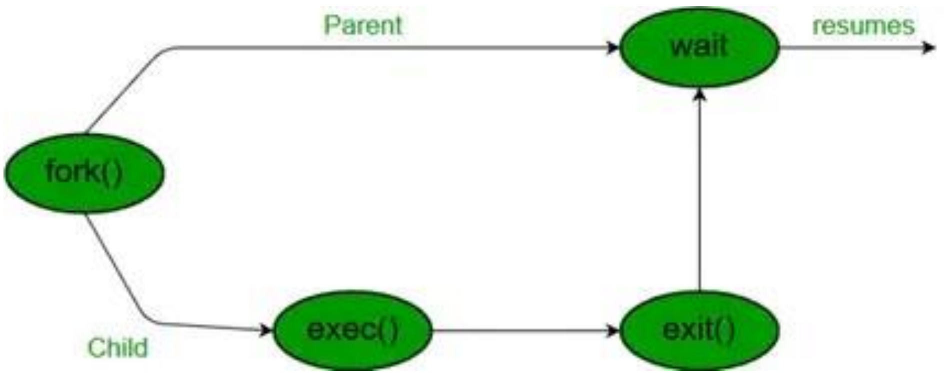
Result:

Thus fork system call program was executed Successfully

Wait ( )

Theory:
A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent continues its execution after wait system call instruction.
The child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate



- If any process has more than one child process, then after calling wait(), the parent process has to be in a wait state if no child terminates.
- If only one child process is terminated, then return a wait() returns the process ID of the terminated child process.
- If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process.
- When wait() returns they also define exit status (which tells our, a process why terminated) via a pointer, If status are not **NULL**.

If any process has no child process then wait() returns immediately "-1".

Program:

```
// C program to demonstrate working of wait()

#include<stdio.h>

#include<sys/wait.h>

#include<unistd.h>


int main()

{

   if (fork()== 0)

       printf("HC: hello from child\n");

   else

   {

       printf("HP: hello from parent\n");

       wait(NULL);

       printf("CT: child has terminated\n");

   }


   printf("Bye\n");

   return 0;

}
```

Output

```
HC: hello from child

HP: hello from parent

CT: child has terminated

    (or)

HP: hello from parent

HC: hello from child

CT: child has terminated     // this sentence does

                            // not print before HC

                            // because of wait.
```

Exec( )

- The exec family of functions replaces the currently running process with a new process.
- It can be used to run a C program by using another C program.
- It comes under the header file **unistd.h.**

Program:

```
// parent.c: the parent program

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/wait.h>

int main (int argc, char **argv)

{

   int i = 0;
```

```
    long sum;

    int pid;

    int status, ret;

    printf ("Parent: Hello, World!\n");

    pid = fork ();

    if (pid == 0) {

        // I am the child

        execvp ("./child", NULL);

    }


// I am the parent

    printf ("Parent: Waiting for Child to complete.\n");

    if ((ret = waitpid (pid, &status, 0)) == -1)

         printf ("parent:error\n");

    if (ret == pid)

        printf ("Parent: Child process waited for.\n");

}
```

Output:

```
gcc parent.c -o parent   //compiling parent.c

./parent                 // running parent.c
```

<u>Experiment No: 2</u>

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

Aim: To Write a Program Using C to Implement the a) First Come First Serve Scheduling b) Shortest Job First c) Round Robin d) Priority Scheduling &to find Turnaround time waiting time.

Theory:

- Suppose there are 3 processes in the ready list. Further, assume that they had entered into the ready list in the order P0, P1, P2.

i	T(Pi)
0	24
1	3
2	3

Gantt (Gantl Chart):

P0	P1	P2

0 24 27 30

Turn Around Time:

TRND(P0) =T(P0) =24

TRND(P1) = T(P1)+ TRND(P0) = 3+24 = 27

TRND(P2) =T(P2)+ TRND(P1) = 3+27 = 30

Average Turn Around Time:

TRND =(24+27+30)/3=27

Waiting Time:

W(P0)=0

W(P1)= TRND(P0)=24 or W(P1)= W(P0)+burst time of P0 = 0+24=24

W(P2)= TRND(P1)=27

Average Waiting Time:-

(0+24+27)/3=17

- Consider the following processes arrive for execution at the times indicated.
- Each process will run for the amount of time listed.
- In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	**Arrival Time**	**Burst Time**
P1	0.0	8
P2	0.4	4
P3	1.0	1

Algorithm:

- STEP 1- START
- STEP 2- To declare the n, bt, wt, tat, avwt, i, and j.
- STEP 3- Print the total number of process in n. Enter process Burst Time
- STEP 4- The bt and wt in for loop ‘i ‘value is 0, i<n and i increment.
- STEP 5- Print the Process , Burst time, Waiting time and Turnaround Time
- STEP 6- wt in for loop ‘j ‘value is 0, j<i and j increment.
- STEP 7- tat equal to bt add wt, avwt+=wt[i], avtat+=tat[i], Print the tat, bt and wt.
- STEP 8- avwt/=i, avtat/=I . And print the average waiting time and average turnaround time.

STEP 9- STOP

Program:

```
#include<stdio.h>

int main()

{

int n,bt[10],wt[10],tat[10],avwt=0,avtat=0,i,j;

printf("Enter total number of processes");

scanf("%d",&n);

printf("\nEnter Process Burst Time\n");

for(i=0;i<n;i++)

{

printf("%d",i+1);

scanf("%d",&bt[i]);

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j++)

wt[i]+=bt[j];

}

printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

for(i=0;i<n;i++)

{
```

```
tat[i]=bt[i]+wt[i];

avwt+=wt[i];

avtat+=tat[i];

printf("\n%d\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);

}

avwt/=i;

avtat/=i;

printf("\n\nAverage Waiting Time:%d",avwt);

printf("\nAverage Turnaround Time:%d",avtat);

return 0;

}
```

Execution:

Input

```
3 4 3 5
```

Output

```
Enter Total Number of processes:

Enter Process Burst Time:

Enter P1:

Enter P2:

Enter P3:
```

```
Process       Burst Time    Waiting Time    Turnaround Time

P1                 4               0                 4

P2                 3               4                 7

P3                 5               7                 12



Avgerage Waiting Time:3.66

Avg Turn around Time :7.66
```

Result:

Thus first come first serve scheduling program was executed Successfully.

a) SJF

Theory:
In the following example assume no other jobs arrive during the servicing of all jobs in the ready list. Assume there are 4 jobs.

i	T(Pi)
0	5
1	10
2	8
3	3

Since the service time of P3 is 3, which is the shortest, P3 is scheduled first and then P0 is scheduled next and so on.

Gantt Chart:

P3	P0	P2	P1

0 3 8 16 28

Turn Around Time:

TRND(P0)=T(P0)+T(P3)=5+3=8

TRND(P1)=T(P1)+ T(P0)+T(P3)+T(P2)=10+5+3+5=26

TRND(P2)=T(P2)+ T(P0)+T(P3)=8+5+3=16

TRND(P3)= T(P3)= 3

Average Turn Around Time:

TRND =(8+26+16+3)/4=13

Waiting Time:

W(P0)=3

W(P1)=16

W(P2)= 8

W(P3)= 0

Average Waiting Time:

W=(3+16+8+0)/4=6.75

Algorithm:

- STEP 1- START
- STEP 2- Declare the integer bt, p, wt, ta, i, j, n, tot=0, pos and temp
- STEP 3- Declare to float average waiting time and average turnaround time.
- STEP 4-Print the number of process in n and Burst time.
- STEP 5- The bt in for loop 'i 'value is 0, i<n and i increment in p[i]+i.
- STEP 6- pos=i, The bt in for loop 'j 'value is i+1, j<n and j increment.
- STEP 7- if bt[j] less than bt[pos], pos=j and create the temporary file and equal to the bt[i], bt[i] equal to bt[pos], bt[pos]equal to temporary, temporary equal to p[i], so p[i] equal to p[pos] and p[pos] equal their temporary file
- STEP 8- wt[0] ,the wt in for loop 'i 'value is 0, i<n and i increment, becomes wt[i], In for loop 'j 'value is i+1, j<n and j increment wt[i] add or equal to bt[j], tot add or equal to wt[i].
- STEP 9- In float average waiting time equal to total divided be n value, And the value of total is 0.
- STEP 10- Print the value process burst time waiting time and turnaround time.
- STEP 11- tat[i] equal to bt[i] add the wt[i], tot add or equal to tat[i],and print the value p[i],bt[i],wt[i] andtat[i].
- STEP 12- In float average turnaround time equal to total divided be n value, and print the average waiting time and average turn around time.
- STEP 13- STOP.

Program:

```
#include<stdio.h>

void main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,tot=0,pos,temp;

float avgwt,avgtat;

printf("Enter number of process:");

scanf("%d",&n);

printf("\nEnter Burst Time:\n");

for(i=0;i<n;i++)

{

printf("p%d:",i+1);

scanf("%d",&bt[i]);

p[i]=i+1;

}

for(i=0;i<n;i++)

{

pos=i;

for(j=i+1;j<n;j++)

{

if(bt[j]<bt[pos])

pos=j;
}
temp=bt[i];
bt[i]=bt[pos];
bt[pos]=temp;
temp=p[i];
p[i]=p[pos];
p[pos]=temp;
}
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
tot+=wt[i];
}
avgwt=(float)tot/n;
tot=0;
printf("\nProcess\t    Burst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];
```

```
tot+=tat[i];

printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avgtat=(float)tot/n;

printf("\n\nAverage Waiting Time=%f",avgwt);

printf("\nAverage Turnaround Time=%f\n",avgtat);

}
```

Execution

Input

```
4 5 3 9 2
```

Output

```
Enter Total Number of processes:

Enter Process Burst Time:

Enter P1:

Enter P2:

Enter P3:

Process     Burst Time     Waiting Time     Turnaround Time

 P4              2               0                2

 P2              3               2                5

 P1              5               5                10

 P3              9               10               19
```

Average Waiting Time= 4.25

Avg Turn Around Time= 9.00

Result:

Thus Shortest job first scheduling program was executed Successfully.

Round Robin Scheduling

Theory:

- Suppose the ready list contains the processes as shown in table and time quantum is 4 with a negligible amount of time for context switching.

i	**T(Pi)**
0	**24**
1	**3**
2	**3**

Gantt Chart:

P0	**P1**	**P2**	**P0**	**P0**	**P0**	**P0**	**P0**

0 4 7 10 14 18 22 26 30

Turn Around Time:

TRND(P0)=30

TRND(P1)=7

TRND(P2)=10

Average Turn Around Time:

TRND =(30+7+10)/3=15.66

Waiting Time:

W(P0)=26-(3+3+4+4+4+4) = 26-20 =6

W(P1)=4

W(P2)=7

Average Waiting Time:

W=(6+4+7)/3=5.66667

Algorithm:

- STEP-1: The queue structure in ready queue is of First In First Out (FIFO) type.
- STEP-2: A fixed time is allotted to every process that arrives in the queue. This fixed time is known as time slice or time quantum.
- STEP-3: The first process that arrives is selected and sent to the processor for execution. If it is not able to complete its execution within the time quantum provided, then an interrupt is generated using an automated timer.
- STEP-4: The process is then stopped and is sent back at the end of the queue. However, the state is saved and context is thereby stored in memory. This helps the process to resume from the point where it was interrupted.
- STEP-5: The scheduler selects another process from the ready queue and dispatches it to the processor for its execution. It is executed until the time Quantum does not exceed.
- STEP-6: The same steps are repeated until all the process are finished.
- The round-robin algorithm is simple and the overhead in decision making is very low.
- It is the best scheduling algorithm for achieving better and evenly distributed response time.

Program:

```
#include<stdio.h>

int main()

#define max 10

{

int i,at[max],bt[max],temp[max],NOP,sum=0,count=0,y,quant,wt=0,tat=0;

float avg_wt, avg_tat;

printf(" Enter The Total number of Process: ");

scanf("%d", &NOP);

y = NOP;

for(i=0; i<NOP; i++)

{

printf("\n Enter the Arrival & Burst time of P[%d]: ", i+1);
```

```
scanf("%d%d", &at[i],&bt[i]);

temp[i] = bt[i];

}

printf("\n Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process BT \tTAT \t WT ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;
printf("\n P[%d] \t %d \t %d \t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}  }

avg_wt = wt * 1.0/NOP;

avg_tat = tat * 1.0/NOP;

printf("\n Average WT : %.2f", avg_tat);

printf("\n Average TAT: %.2f", avg_wt);

}
```

Execution:

Input

```
3 0 4 1 6 2 9 3
```

Output

```
 Enter The Total number of Process:

 Enter the Arrival & Burst time of P[1]:

 Enter the Arrival & Burst time of P[2]:

 Enter the Arrival & Burst time of P[3]:

 Enter the Time Quantum for the process:

 Process BT        TAT      WT

 P[1]       4        10       6

 P[2]       6        12       6

 P[3]       9        17       8

 Average WT : 13.00

 Average TAT: 6.67
```

Result:

Thus round robin scheduling program was executed Successfully.

Priority Scheduling

Theory:
Suppose the ready list contains the processes as shown in the table.

i	TP(i)	Priority
0	24	2
1	3	1
2	3	3

Gantt Chart:

P1	**P0**	**P2**

0 3 27 30

Turn Around Time:

TRND(P0)=27

TRND(P1)=3

TRND(P2)=30

Average Turn-Around Time:

TRND =(30+3+27)/3=20

Waiting Time:

W(P0)=3

W(P1)=0

W(P2)=27

Average Waiting Time:

W=(3+0+27)/3=10

Program:

```
#include<stdio.h>

#define max 10

int main()

{

int i,j,n,bt[max],p[max],wt[max],tat[max],pr[max],total=0,pos,temp;

float avg_wt,avg_tat;

printf("Enter Total Number of Process:");

scanf("%d",&n);

printf("\nEnter Burst Time and Priority For ");

for(i=0;i<n;i++)

{

printf("\nEnter Process %d: ",i+1);

scanf("%d",&bt[i]);

scanf("%d",&pr[i]);

p[i]=i+1;

}

for(i=0;i<n;i++)

{

pos=i;

for(j=i+1;j<n;j++)

{

if(pr[j]<pr[pos])
pos=j;

}

temp=pr[i];

pr[i]=pr[pos];

pr[pos]=temp;

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j++)

wt[i]+=bt[j];

total+=wt[i];

}

avg_wt=total/n;

total=0;
```

```
printf("\n\nProcess\t\tBT\t\tWT\t\tTAT");

for(i=0;i<n;i++)

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("\n P%d\t\t%d\t\t%d\t\t%d",p[i],bt[i],wt[i],tat[i]);

   }

avg_tat=total/n;

printf("\n\nAverage Waiting Time = %.2f",avg_wt);

printf("\nAvg Turn Around Time = %.2f\n",avg_tat);

return 0;

}
```

Execution:

Input

```
4 3 2 5 1 7 5 9 4
```

Output

```
Enter Total Number of Process:

Enter Burst Time and Priority For

Enter Process 1:

Enter Process 2:

Enter Process 3:

Enter Process 4:
```

```
Process          BT               WT               TAT

 P2              5                0                5

 P1              3                5                8

 P4              9                8                17

 P3              7                17               24


Average Waiting Time = 7.00

Avg Turn Around Time = 13.00
```

Result:

Thus priority scheduling program was executed Successfully.

Experiment No: 3

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

Aim: To Develop a C program to simulate producer-consumer problem using semaphores

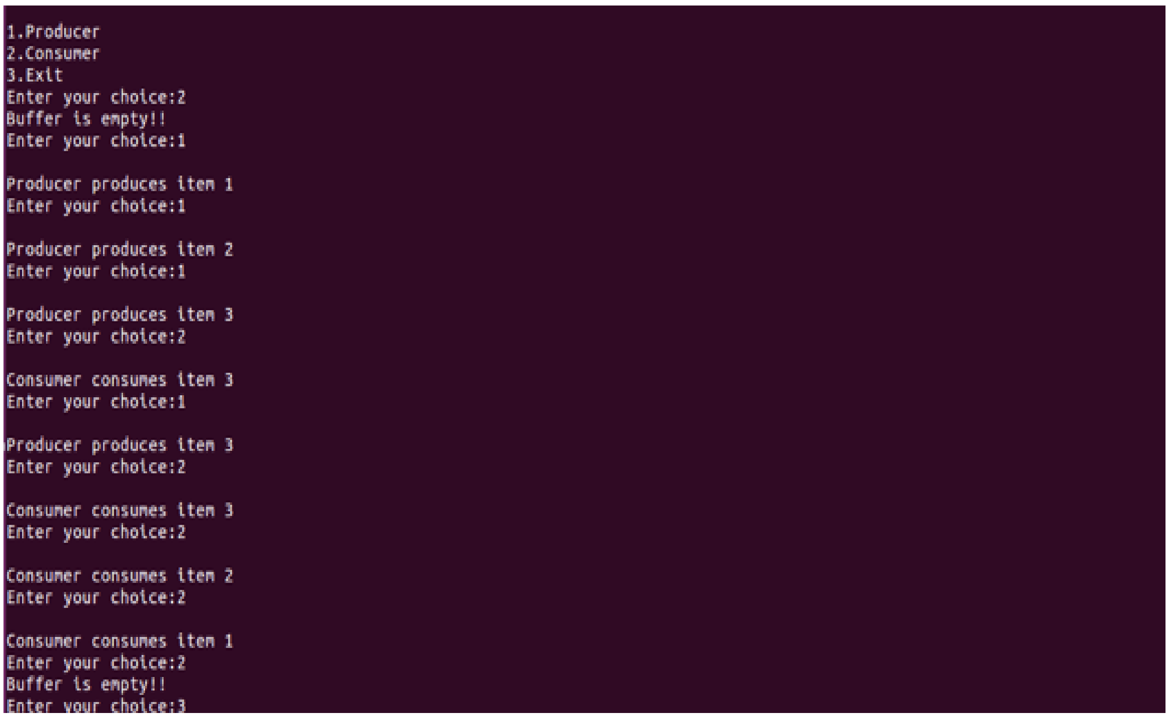
Theory: Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the produced.-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there 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.

Program:

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

// Initialize a mutex to 1
int mutex = 1;

// Number of full slots as 0
int full = 0;

// Number of empty slots as size
// of buffer
int empty = 10, x = 0;

// Function to produce an item and
// add it to the buffer
void producer()
{
   // Decrease mutex value by 1
   --mutex;

   // Increase the number of full
   // slots by 1
   ++full;

   // Decrease the number of empty
   // slots by 1
   --empty;

   // Item produced
   x++;
   printf("\nProducer produces"
            "item %d",
            x);

    // Increase mutex value by 1
    ++mutex;
}

// Function to consume an item and
// remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes "
            "item %d",
            x);
    x--;

    // Increase mutex value by 1
    ++mutex;
}

// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
            "\n2. Press 2 for Consumer"
            "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can  give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
#pragma omp critical

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

        // Switch Cases
        switch (n) {
        case 1:

            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1)
                && (empty != 0)) {
                producer();
            }

            // Otherwise, print buffer
            // is full
            else {
                printf("Buffer is full!");
            }
            break;

        case 2:

            // If mutex is 1 and full
            // is non-zero, then it is
            // possible to consume
            if ((mutex == 1)
                && (full != 0)) {
                consumer();
            }

            // Otherwise, print Buffer
            // is empty
            else {
                printf("Buffer is empty!");
            }
            break;

        // Exit Condition
        case 3:
            exit(0);
            break;
        }
    }
}
```

Output:

1.
Produce
2.
Consume
3.
Exit
Enter your choice:
Buffer is Empty



```
1.Producer
2.Consumer
3.Exit
Enter your choice:2
Buffer is empty!!
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:2

Consumer consumes item 3
Enter your choice:1

Producer produces item 3
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
```

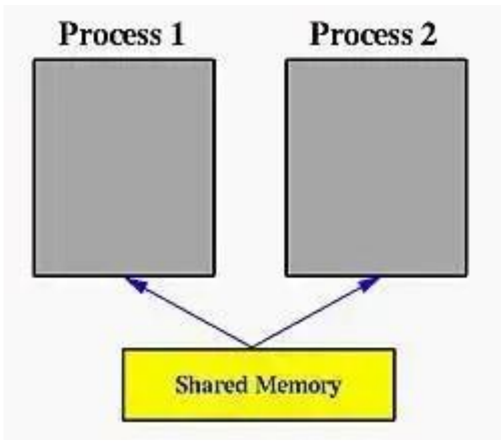
<u>Experiment No</u>: <u>4</u>

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.

Aim: 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.

Theory:

- In the discussion of the **fork()** system call, we mentioned that a parent and their children have separate address spaces.
- While this would provide a more secure way of executing parent and children processes (because they will not interfere with each other), they shared nothing and have no way to communicate with each other.
- ***Shared memory*** is an extra piece of memory that is *attached* to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it.
- Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces.
- The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space.
- In some sense, the original address spaces is "extended" by attaching this shared memory.



- Shared memory is a feature supported by UNIX System V, including Linux, SunOS, and Solaris.
- One process must explicitly ask for an area, using a ***key***, to be shared by other processes. This process will be called the *server*.

- All other processes, the *clients*, that know the shared area can access it. However, there is no protection to shared memory, and any process that knows it can access it freely.
- To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be set up.
- A shared memory segment is identified by a unique integer, the ***shared memory ID***.
- The shared memory itself is described by a structure of type **shmid_ds** in the header file sys/shm.h. To use this file files **sys/types.h** and **sys/ipc.h** must be included.

Algorithm:

1. Create the pipe and create the process.
2. Get the input in the main process and pass the output to the child process using pipe.
3. Perform the operation given in the child process and print the output.
4. Stop the program.

Program:

```
#include<stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>



int main()

{

if(fork()==0)

{

key_t key = ftok("shmfile",65);

int shmid = shmget(key,1024,0666|IPC_CREAT);
char str = (char) shmat(shmid,(void*)0,0);

printf("Write Data : \n");

scanf("%s",str);

printf("Child process written in memory: %s\n",str);

shmdt(str);

}

else

{

wait(0);

key_t key = ftok("shmfile",65);

int shmid = shmget(key,1024,0666|IPC_CREAT);

char str = (char) shmat(shmid,(void*)0,0);

printf("Parent reads from memory        : %s\n",str);

shmdt(str);

shmctl(shmid,IPC_RMID,NULL);

}

return 0;

}
```

Result:

Thus the implement inter process communication program was executed Successful

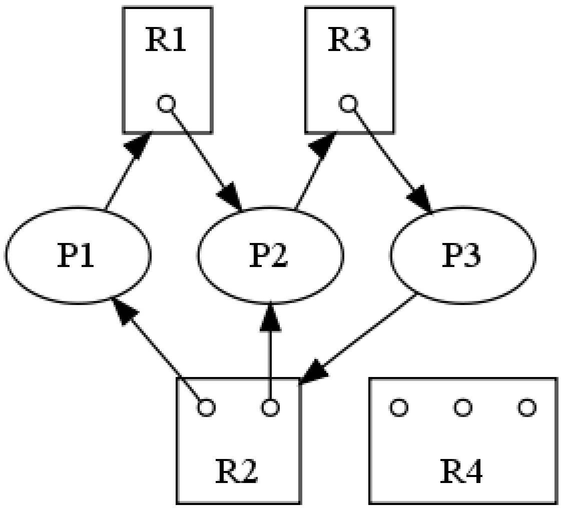
Experiment No: 5

Develop a C program to simulate Bankers Algorithm for Dead Lock Avoidance.

Aim: To Develop a C program to simulate Bankers Algorithm for Dead Lock Avoidance.

Theory:

- The Banker's Algorithm was designed and developed by a Dutch Computer Scientist, Edsger Djikstra. The Banker's Algorithm is a Resource Allocation and a Deadlock Avoidance Algorithm.
- This algorithm takes analogy of an actual bank where clients request to withdraw cash.
- The Banking Authorities have some data according to which the cash is lent to the client.
- The Banker cannot give more cash than the client's request and the total cash available in the bank.



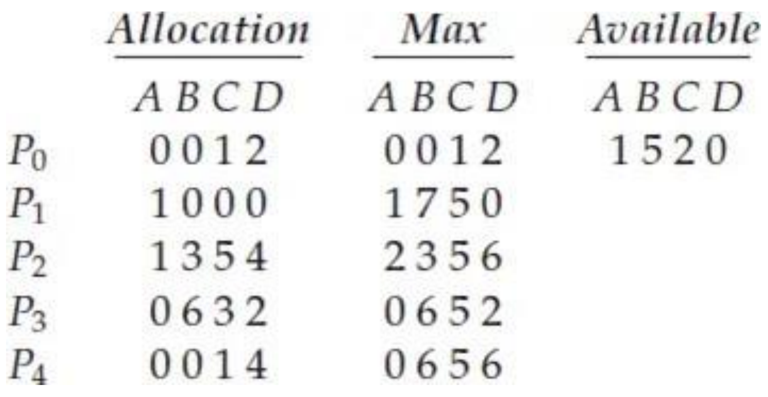
The Banker's Algorithm is divided into two parts:

1. **Safety Test Algorithm:** This algorithm checks the current state of the system to maintain its Safe State.
2. **Resource Request Handling Algorithm:** This algorithm verifies if the requested resources, after their allocation to the processes affects the Safe State of the System. If it does, then the request of the process for the resource is denied, thereby maintaining the Safe State.

A State is considered to be Safe if it is possible for all the Processes to Complete its Execution without causing any Deadlocks. An Unsafe State is the one in which the Processes cannot complete its execution.

Exercise:

Consider the following snapshot of a system:



	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Algorithm:

1. The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources
2. Then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.
3. Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

int need[100][100],avai[100];

struct process{

int pc;

bool status;

}pno[100];
```

```
bool check(int i,int m)
{
int j;
for(j=0;j<m;j++)
{
if(need[i][j]>avai[j])
return 0;
}
return 1;
}
void disp(int n)
{
int i;
printf("SYSTEM IN SAFE STATE\nSAFE SEQUENCE = ");
for(i=0;i<n;i++)
{
printf("P%d ",pno[i].pc);
}
}
void banker(int alloc[][100],int max[][100],int n,int m)
{
int i,j,ck=0,ic=0,flag=0;
```

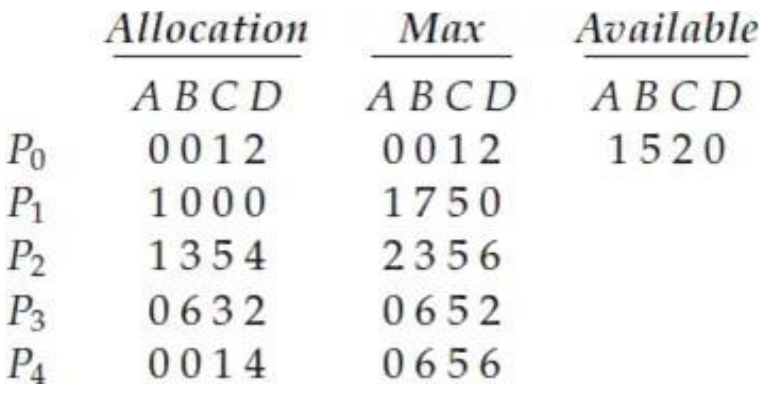
```
bool checker=0;

printf("NEED MATRIX :\n");

for(i=0;i<n;i++)

{

for(j=0;j<m;j++)

{

need[i][j]=max[i][j]-alloc[i][j];

printf("%d ",need[i][j]);

pno[i].status=0;

}

printf("\n");

}

while(ck<n)

{

if(check(ic,m)==1&&pno[ic].status==0)

{

int ac=0;

pno[ck].pc=ic;

ck++;

pno[ic].status=1;

checker=1;

for(ac=0;ac<m;ac++)
{

avai[ac]=avai[ac]+alloc[ic][ac];

}

}

if(ic==n-1&&checker==0)

{

printf("SYSTEM UNSAFE\n");

flag=1;

ck=n;

}

if(ic==n-1)

{

ic=-1;

checker =0;

}

ic++;

}

if(flag==0)

disp(n);


}
```

```
int main()

{

int alloc[100][100],max[100][100],n,m,i,j;

printf("ENTER THE NO PROCESS AND RESOURCES : ");

scanf("%d%d",&n,&m);

printf("\nENTER ALLOCATION MATRIX : \n");

for(i=0;i<n;i++)

for(j=0;j<m;j++)

scanf("%d",&alloc[i][j]);

printf("ENTER MAX MATRIX : \n");

for(i=0;i<n;i++)

for(j=0;j<m;j++)

scanf("%d",&max[i][j]);

printf("ENTER AVAILABLE :\n");

for(i=0;i<m;i++)

scanf("%d",&avai[i]);

banker(alloc,max,n,m);

return 0;

}
```

Execution:

Input:

```
5 4 0 0 1 2 1 0 0 0 1 3 5 4 0 6 3 2 0 0 1 4 0 0 1 2 1 7 5 0 2 3 5 6 0 6 5 2 0 6 5 6 1 5 2 0
```



	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Output:

```
ENTER THE NO PROCESS AND RESOURCES :

ENTER ALLOCATION MATRIX :

ENTER MAX MATRIX :

ENTER AVAILABLE :

NEED MATRIX :

0 0 0 0

0 7 5 0

1 0 0 2

0 0 2 0

0 6 4 2

SYSTEM IN SAFE STATE

SAFE SEQUENCE = P0 P2 P3 P4 P1
```

Experiment No: 6

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

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

a)Worst fit

Algorithm

- **Worst fit Algorithm**
- Get no. of Processes and no. of blocks.
- After that get the size of each block and process requests.
- Now allocate processes
- if(block size >= process size)
- else
- Display the processes with the blocks that are allocated to a respective process.
- Stop.

Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

struct p{

int acum[100];

int jp[100];

}st;

int main()

{

int n,m,i,count=0,j,pn[100];

int p[100],size[100];
```

```
bool flag[100];

printf("ENTER THE NO PROCESS AND MEMORY :\n ");

scanf("%d%d",&n,&m);

printf("ENTER THE SIZE OF PROCESS \n");

for(i=0;i<n;i++)

scanf("%d",&p[i]);

printf("ENTER THE SIZE OF MEMORY PARTION \n");

for(i=0;i<m;i++)

{

scanf("%d",&size[i]);

flag[i]=0;

int k;

st.acum[i]=size[i];

st.jp[i]=i;

for(k=i;k>0;k--)

{

if(st.acum[k]<=st.acum[k-1])

{

int temp=st.acum[k];

st.acum[k]=st.acum[k-1];

st.acum[k-1]=temp;

temp=st.jp[k];
st.jp[k]=st.jp[k-1];

st.jp[k-1]=temp;

}

}

}

int x=m-1;

for(i=0;i<n;i++)

{

if(p[i]<=st.acum[x]&&flag[st.jp[x]]==0)

{

flag[st.jp[x]]=true;

pn[st.jp[x]]=i;

x--;

count++;

}

}

printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);

printf("MEMORY\tPROCESS\n");

for(i=0;i<m;i++ )

{

if(flag[i]==1)

{
```

```
printf("%d <-->%d\n",size[i],p[pn[i]]);

}

else

printf("%d\tMEMORY NOT ALLOCATED\n",size[i]);

}

return 0;

}
```

Execution:

Input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMORY :

ENTER THE SIZE OF PROCESS

ENTER THE SIZE OF MEMORY PARTION

NO OF PROCESS CAN ACOMADATE :3


MEMORY          PROCESS

100        MEMORY NOT ALLOCATED

500 <-->417

200        MEMORY NOT ALLOCATED

300 <-->112

600 <-->212
```

Result:

Thus Memory Management Scheme using the Worst Fit program was executed Successfully.

b)Best Fit

Algorithm:

1. **Best Fit Algorithm**
2. Get no. of Processes and no. of blocks.
3. After that get the size of each block and process requests.
4. Then select the best memory block that can be allocated using the above definition.
5. Display the processes with the blocks that are allocated to a respective process.
6. Value of Fragmentation is optional to display to keep track of wasted memory.
7. Stop.

Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

struct p{

int acum[100];

int jp[100];

}st;

int main()

{

int n,m,i,count=0,j,pn[100];

int p[100],size[100];

bool flag[100];

printf("ENTER THE NO PROCESS AND MEMORY :\n ");
```

```
scanf("%d%d",&n,&m);

printf("ENTER THE SIZE OF PROCESS \n");

for(i=0;i<n;i++)

scanf("%d",&p[i]);

printf("ENTER THE SIZE OF MEMORY PARTION \n");

for(i=0;i<m;i++)

{

scanf("%d",&size[i]);

flag[i]=0;

}

for(i=0;i<n;i++)

{

int ic=0,in=0;

for(j=0;j<m;j++)

{


if(p[i]<=size[j]&&flag[j]==0)

{


int k;

st.acum[in]=size[j];

st.jp[in]=j;
in++;

ic++;

for(k=ic-1;k>0;k--)

{

if(st.acum[k]<=st.acum[k-1])

{

int temp=st.acum[k];

st.acum[k]=st.acum[k-1];

st.acum[k-1]=temp;

temp=st.jp[k];

st.jp[k]=st.jp[k-1];

st.jp[k-1]=temp;

}

}

}

}

if(ic>0)

{

j=st.jp[0];

flag[j]=true;

pn[j]=i;

count++;
```

```
}

}

printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);

printf("MEMORY\tPROCESS\n");

for(i=0;i<m;i++ )

{

if(flag[i]==1)

{

printf("%d <-->%d\n",size[i],p[pn[i]]);

}

else

printf("%d\tMEMORY NOT ALLOCATED\n",size[i]);

}

return 0;

}
```

Execution:

Input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMORY :

ENTER THE SIZE OF PROCESS

ENTER THE SIZE OF MEMORY PARTION

NO OF PROCESS CAN ACOMADATE :4

MEMORY          PROCESS

100        MEMORY NOT ALLOCATED

500 <-->417

200 <-->112

300 <-->212

600 <-->426
```

Result:

Thus Memory Management Scheme using the Best Fit program was executed Successfully.

b) First Fit:

Algorithm:

1. **First Fit Algorithm**
2. Get no. of Processes and no. of blocks.
3. After that get the size of each block and process requests.
4. Now allocate processes
5. if(block size >= process size)
6. else
7. Display the processes with the blocks that are allocated to a respective process.
8. Stop.

Program

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

int main()

{

int n,m,i,count=0,j,pn[100];

int p[100],size[100];

bool flag[100];

printf("ENTER THE NO PROCESS AND MEMOR :\n ");

scanf("%d%d",&n,&m);

printf("ENTER THE SIZE OF PROCESS \n");

for(i=0;i<n;i++)

scanf("%d",&p[i]);

printf("ENTER THE SIZE OF MEMOR PARTION \n\n");

for(i=0;i<m;i++)

{

scanf("%d",&size[i]);

flag[i]=0;

}

for(i=0;i<n;i++)

{
```

```
for(j=0;j<m;j++)

{

if(p[i]<=size[j]&&flag[j]==0)

{

flag[j]=true;

pn[j]=i;

count++;

j=j+m;

}

}

}

printf("NO OF PROCESS CAN ACOMADATE :%d\n\n",count);

printf("MEMOR\tPROCESS\n");

for(i=0;i<m;i++ )

{

if(flag[i]==1)

{

printf("%d <-->%d\n",size[i],p[pn[i]]);

}

else

printf("%d\tMEMOR NOT ALLOCATED\n",size[i]);

}
```

```
return 0;

}
```

Execution:

input:

```
4 5 212 417 112 426 100 500 200 300 600
```

Output:

```
ENTER THE NO PROCESS AND MEMOR :

 ENTER THE SIZE OF PROCESS

ENTER THE SIZE OF MEMOR PARTION

NO OF PROCESS CAN ACOMADATE :3

MEMORPROCESS

100        MEMOR NOT ALLOCATED

500 <-->212

200 <-->112

300        MEMOR NOT ALLOCATED

600 <-->417
```

Experiment No: 7

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

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

a) FIFO

Theory:

- Page replacement algorithms are used to decide what pages to page out when a page needs to be allocated.
- This happens when a page fault occurs and a free page cannot be used to satisfy the allocation.

FIFO:

- "Replace *the page that had not been used* for a longer sequence of time".
- The frames are empty in the beginning and initially no page fault occurs so it is set to zero. When a page fault occurs the page reference sting is brought into the memory.
- The operating system keeps track of all pages in the memory, thereby keeping track of the page that had not been used for longer sequence of time.
- If the page in the page reference string is not in memory, the page fault is incremented and the page that had not been used for a longer sequence of time is replaced. If the page in the page reference string is in the memory take the next page without calculating the next page.
- Take the next page in the page reference string and check if the page is already present in the memory or not.
- Repeat the process until all pages are referred and calculate the page fault for all those pages in the page references string for the number of available frames.

Algorithm:

- Step 1. Start to traverse the pages.
- Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
- Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
- Step 4. If the current page is present in the memory, do nothing.
- Step 5. Else, pop the topmost page from the queue as it was inserted first.
- Step 6. Replace the topmost page with the current page from the string.
- Step 7. Increment the page faults.
- Step 8. Stop

Program:

```
#include <stdio.h>

#include<stdbool.h>

int rs[100],count =0,f[10];

bool dataavi(int n)

{

int i;

for(i=0;i<n;i++)

{

if(rs[count]==f[i])

return 1;

}

return 0;

}



void fifo(int n,int m)

{

int fs=0,i=0,j=0,kom=0;

while(count<m)

{

if(count<n)

{
```

```
f[count]=rs[count++];

fs++;

kom=1;

}

else

{

if(dataavi(n))

{

count++;

kom=0;

}

else

{

kom=1;

f[j]=rs[count++];

fs++;

j++;

if(j>=n)

j=0;

}

}

if(kom==1)
```

```
printf(" Page Fault :");

else

printf("            :");

for(i=0;i<n;i++)

printf(" %d",f[i]);

printf("\n");

}

printf("\n\nPAGE FAULTS = %d",fs);

}

int main()

{

int n ,m,i;

printf("ENTER THE NO OF FRAME AND REFERENCE STREAM\n");

scanf ("%d%d",&n,&m);

printf ("ENTER THE REFERENCE STREAM\n");

for(i=0;i<m;i++)

scanf("%d",&rs[i]);

fifo(n,m);

return 0;

}
```

Execution: Input:

```
3 18 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2
```

Output:

```
ENTER THE NO OF FRAME AND REFERENCE STREAM

ENTER THE REFERENCE STREAM

 Page Fault : 1 0 0

 Page Fault : 1 2 0

 Page Fault : 1 2 3

 Page Fault : 4 2 3

            : 4 2 3

 Page Fault : 4 1 3

 Page Fault : 4 1 5

 Page Fault : 6 1 5

 Page Fault : 6 2 5

 Page Fault : 6 2 1

            : 6 2 1

 Page Fault : 3 2 1

 Page Fault : 3 7 1

 Page Fault : 3 7 6

            : 3 7 6

 Page Fault : 2 7 6

 Page Fault : 2 1 6

            : 2 1 6
```

Page Fault : 2 1 3

Page Fault : 6 1 3

PAGE FAULTS = 16

b) LRU

Theory:

- Page replacement algorithms are used to decide what pages to page out when a page needs to be allocated.
- This happens when a page fault occurs and a free page cannot be used to satisfy the allocation

LRU:

- "Replace *the page that had not been used* for a longer sequence of time".
- The frames are empty in the beginning and initially no page fault occurs so it is set to zero. When a page fault occurs the page reference sting is brought into the memory.
- The operating system keeps track of all pages in the memory, thereby keeping track of the page that had not been used for longer sequence of time.
- If the page in the page reference string is not in memory, the page fault is incremented and the page that had not been used for a longer sequence of time is replaced. If the page in the page reference string is in the memory take the next page without calculating the next page.
- Take the next page in the page reference string and check if the page is already present in the memory or not.
- Repeat the process until all pages are referred and calculate the page fault for all those pages in the page references string for the number of available frames.

Algorithm:

- 1- Start traversing the pages.
  1. i) If set holds less pages than capacity.
     - a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
     - b) Simultaneously maintain the recent occurred index of each page in a map called indexes.
     - c) Increment page fault
  2. ii) Else If current page is present in set, do nothing.
     - Else
       1. a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
       2. b) Replace the found page with current page.
       3. c) Increment page faults.
       4. d) Update index of current page.
- 2 – Return page faults.

Program:

```
#include <stdio.h>

#include<stdbool.h>

int rs[100],count =0;

struct max{

int f[10];

bool lu[10];

}s;

struct min{

int buf[10];

int mat[10];

}nat;

bool dataavi(int n)

{

int i;

for(i=0;i<n;i++)

{

if(rs[count]==s.f[i])

return 1;

}


return 0;
```

```
}

int check(int n,int m)

{

int i,j=0;

for(j=0;j<n;j++)

{

int x=0;



for(i=count+1;i<m;i++)

{



if(s.f[j]==rs[i])

{

nat.buf[j]=i;

nat.mat[j]=j;

i=m+1;

x=1;

}

}

if(x==0&&s.lu[j]==0)

return j;

}
```

```
for (i = 0; i < n-1; i++)

{

for (j = 0; j < n-i-1; j++)

{

if (nat.buf[j] < nat.buf[j+1])

{

int temp=nat.buf[j];

nat.buf[j]=nat.buf[j+1];

nat.buf[j+1]=temp;

temp=nat.mat[j];

nat.mat[j]=nat.mat[j+1];

nat.mat[j+1]=temp;

}



}}

for(i=0;i<n;i++)

{

if(s.lu[nat.mat[i]]==0)

{

return nat.mat[i];

}
```

```
}

}


void lru (int n,int m)

{

int fs=0,i=0,in=0,kom=0;

for(i=0;i<n;i++)

s.lu[i]=false;

s.lu[n-1]=true;

while(count<m)

{

if(in<n)

{

if(dataavi(n)&&in>0)

{

count++;kom=0;

}

else

{

s.f[in++]=rs[count]; kom=1;

fs++;

count++;
}
}
else
{
if(dataavi(n))
{
count++; kom=0;
}
else
{
int j=check(n,m);
s.f[j]=rs[count++];
fs++; kom=1;
for(i=0;i<n;i++)
s.lu[i]=false;
s.lu[j]=true;
}
}
if(kom==1)
printf(" Page Fault :");
else
printf("            :");
```

```
for(i=0;i<n;i++)

printf(" %d",s.f[i]);

printf("\n");



}

printf("\n\npage faults =%d",fs);

}

int main()

{

int n ,m,i;

printf("ENTER THE NO OF FRAME AND REFERENCE STREAM\n");

scanf ("%d%d",&n,&m);

printf ("ENTER THE REFERENCE STREAM \n");

for(i=0;i<m;i++)

scanf("%d",&rs[i]);

lru (n,m);

return 0;

}
```

Execution:

Input:

```
3 18 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
```

Output:

```
ENTER THE NO OF FRAME AND REFERENCE STREAM

ENTER THE REFERENCE STREAM

 Page Fault : 1 0 0

 Page Fault : 1 2 0

 Page Fault : 1 2 3

 Page Fault : 4 2 3

            : 4 2 3

 Page Fault : 4 2 1

 Page Fault : 5 2 1

 Page Fault : 5 2 6

            : 5 2 6

 Page Fault : 1 2 6

            : 1 2 6

 Page Fault : 1 3 6

 Page Fault : 7 3 6

            : 7 3 6

            : 7 3 6

 Page Fault : 7 3 2

 Page Fault : 1 3 2

            : 1 3 2

            : 1 3 2

 Page Fault : 1 6
```

page faults =13

Result:

Thus LRU Page Replacement Algorithm program was executed Successfully.

Experiment 08 : Simulate following File Organization Techniques

a)Single level directory b) Two level directory

Aim: : Simulate following File Organization Techniques

a)Single level directory b) Two level directory

Theory: The directory contains information about the files, including attributes, location and ownership. Sometimes the directories consisting of subdirectories also. The directory is itself a file, owned by the o.s and accessible by various file management routines. a)Single Level Directories: It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of the all files. Sometimes it is said to be root directory. The following dig. Shows single level directory that contains four files . It has the simplicity and ability to locate files quickly. it is not used in the multi-user system, it is used on small embedded system.

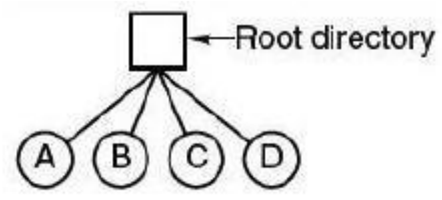


Figure 1. A single-level directory system containing four files

Single level directory

Programe:

```
#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)
{
```

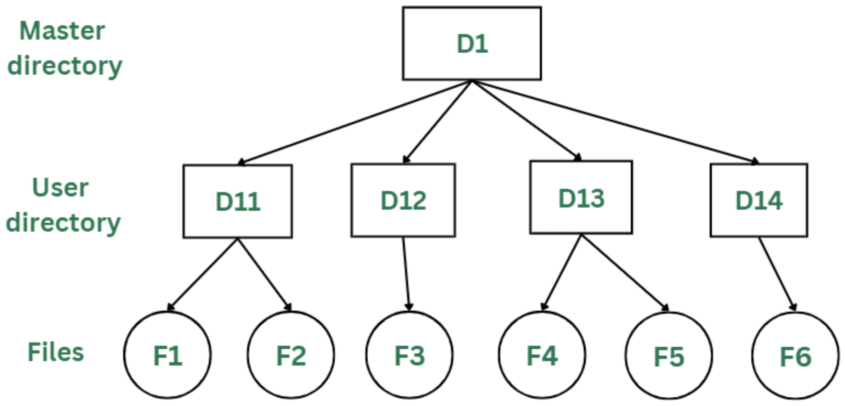
```
printf("\n\n1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter
your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter the name of the file -- ");
scanf("%s",dir.fname[dir.fcnt]);
dir.fcnt++;
break;
case 2: printf("\nEnter the name of the file -- ");
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);
}
}
}
```

Output:

```
Enter the directory name:sss
Enter the number of files:3
Enter file name to be created:aaa
```

b)Two level directory

Theory: Two Level Directory: The problem in single level directory is different users may be accidentally using the same names for their files. To avoid this problem, each user need a private directory. In this way names chosen by one user don‟t interface with names chosen by a different user. The following dig 2-level directory.



Program:

Write C programs to simulate the Two level directory File organization technique

```
#include<string.h>
#include<stdlib.h>
#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];
dcnt=0;
while(1)
{
printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
printf("\n4. Search File\t\t5. Display\t6. Exit\tEnter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter name of directory -- ");
scanf("%s", dir[dcnt].dname);
```

```
dir[dcnt].fcnt=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]);
printf("File created");
break;
}
if(i==dcnt)
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f);
dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter the name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is found ",f);
goto jmp1;
}
}
printf("File %s not found",f);
goto jmp1;
}
}
printf("Directory %s not found",d);
jmp1: break;
case 5: if(dcnt==0)
printf("\nNo Directory's ");
else
{
printf("\nDirectory\tFiles");
for(i=0;i<dcnt;i++)
{
printf("\n%s\t\t",dir[i].dname);
for(k=0;k<dir[i].fcnt;k++)
printf("\t%s",dir[i].fname[k]);
}
}
break;
default:exit(0);
}
}
}
```

Output:

```
Enter the directory name:sss
Enter the number of files:3
Enter file name to be created:aaa
```

Experiment 09: Develop a C program to simulate the Linked file allocation strategies.

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

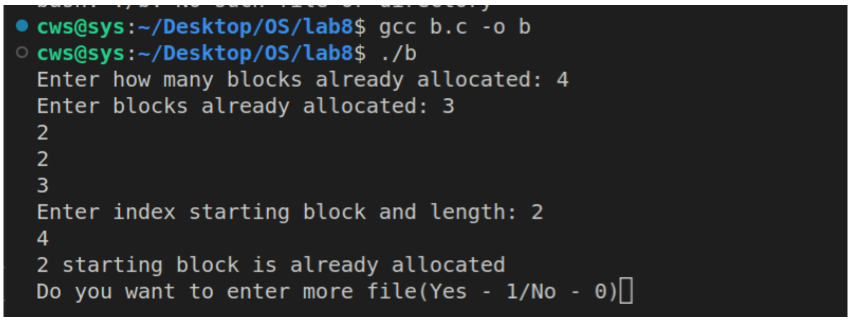
Theory: Linked File Allocation is a Non-contiguous memory allocation method where the file is stored in random memory blocks and each block contains the pointer (or address) of the next memory block as in a linked list. The starting memory block of each file is maintained in a directory and the rest of the file can be traced from that starting block.

The Linked File Allocation is one of the File Allocation Methods in the Operating System. The Linked File Allocation comes under Non-contiguous memory allocation, the other method in the Non-contiguous memory allocation is the Indexed File Allocation, and the last is the Contiguous Memory Allocation.

Program:

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

Out put:



```
cws@sys:~/Desktop/OS/lab8$ gcc b.c -o b
cws@sys:~/Desktop/OS/lab8$ ./b
Enter how many blocks already allocated: 4
Enter blocks already allocated: 3
2
2
3
Enter index starting block and length: 2
4
2 starting block is already allocated
Do you want to enter more file(Yes - 1/No - 0)
```

Experiment 10:

Develop a C program to simulate SCAN disk scheduling algorithm.

Aim: To Develop a C program to simulate SCAN disk scheduling algorithm.

Theory:

- Disk Scheduling is the process of deciding which of the cylinder request is in the ready queue is to be accessed next.
- The access time and the bandwidth can be improved by scheduling the servicing of disk I/O requests in good order.

Access Time:

The access time has two major components: **Seek time and Rotational Latency.**

Seek Time:

Seek time is the time for disk arm to move the heads to the cylinder containing the desired sector.

Rotational Latency:

Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.

Bandwidth:

The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Algorithm:

1. **SCAN Scheduling algorithm** –The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
2. At the other end, the direction of head movement is reversed and servicing continues.
3. The head continuously scans back and forth across the disk.
4. The seek time is calculated.
5. Display the seek time and terminate the program.

Exercise:

- Consider that a disk drive has 5,000 cylinders, numbered 0 to 4,999.
- The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805.
- The queue of pending requests, in FIFO order, is: 2,069, 1,212, 2,296, 2,800, 544, 1,618, 356, 1,523, 4,965, 3681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

1. FCFS
2. SSTF
3. SCAN
4. LOOK
5. C-SCAN
6. C-LOOK

Program:

```
#include<stdio.h>

int request[50];

int SIZE;

int pre;

int head;

int uptrack;

int downtrack;

struct max{

int up;

int down;

}kate[50];

int dist(int a,int b)

{

if(a>b)

return a-b;

return b-a;
```

```
}
void sort(int n)
{
int i,j;
for (i = 0; i < n-1; i++)
  {
for (j = 0; j < n-i-1; j++)
{
if (request[j] > request[j+1])
{
int temp=request[j];
request[j]=request[j+1];
request[j+1]=temp;
}
}
}
j=0;
i=0;
while(request[i]!=head)
{
kate[j].down=request[i];
j++;
```

```
i++;
}
downtrack=j;
i++;
j=0;
while(i<n)
{
kate[j].up=request[i];
j++;
i++;
}
uptrack=j;
}
void scan(int n)
{
int i;
int seekcount=0;
printf("SEEK SEQUENCE = ");
sort(n);
if(pre<head){
for(i=0;i<uptrack;i++)
{
```

```
printf("%d ",head);

seekcount=seekcount+dist(head,kate[i].up);

head=kate[i].up;

}

for(i=downtrack-1;i>0;i--)

{

printf("%d ",head);

seekcount=seekcount+dist(head,kate[i].down);

head=kate[i].down;

}

}

else

{

for(i=downtrack-1;i>=0;i--)

{

printf("%d ",head);

seekcount=seekcount+dist(head,kate[i].down);

head=kate[i].down;

}

for(i=0;i<uptrack-1;i++)

{

printf("%d ",head);
```

```
seekcount=seekcount+dist(head,kate[i].up);

head=kate[i].up;

}

}

printf(" %d\nTOTAL DISTANCE :%d",head,seekcount);

}

int main()

{

int n,i;

printf("ENTER THE DISK SIZE :\n");

scanf("%d",&SIZE);

printf("ENTER THE NO OF REQUEST SEQUENCE :\n");

scanf("%d",&n);

printf("ENTER THE REQUEST SEQUENCE :\n");

for(i=0;i<n;i++)

scanf("%d",&request[i]);

printf("ENTER THE CURRENT HEAD :\n");

scanf("%d",&head);

request[n]=head;

request[n+1]=SIZE-1;

request[n+2]=0;

printf("ENTER THE PRE REQUEST :\n");
```

```
scanf("%d",&pre);

scan(n+3);

}
```

Execution:

Input:

```
5000, 10, 2069 ,1212 , 2296,  2800  ,544, 1618 , 356 ,1523, 4965, 3681, 2150, 1850
```

Output:

```
ENTER THE DISK SIZE : 5000

ENTER THE NO OF REQUEST SEQUENCE :10

ENTER THE REQUEST SEQUENCE :  2069 ,1212 , 2296,  2800  ,544, 1618 , 356 ,1523,
4965, 3681

ENTER THE CURRENT HEAD : 2150

ENTER THE PRE REQUEST : 1850



SEEK SEQUENCE = 2150 ,2296, 2800 ,3681, 4965, 4999, 2069, 1618 ,1523, 1212, 544,
356

TOTAL DISTANCE : 7492
```