1. (a) FCFS (First Come First Serve) CPU Scheduling Algorithm

AIM:

Write a C program to implement the FCFS CPU scheduling.

ALGORITHM:

1. Start the process.
2. Accept the number of processes in the Ready Queue.
3. For each process in the Ready Queue, assign the process id and accept the CPU burst time.
4. Set the waiting of the first process as ‘0’ and its burst time as its turn around time.
5. for each process in the Ready Queue, calculate
    a. Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
    b. Turnaround time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

6. Calculate
    a. Average waiting time = Total waiting Time / Number of process
    b. Average Turnaround time = Total Turnaround Time / Number of process

7. Stop the process.

PROGRAM:

```
#include<stdio.h>
void main()
{
 int bt[10], p[10], wt[10], tat[10], i, j, n, total=0;
 float awt, atat;
 printf("Enter the total number of processes:  ");
 scanf("%d",&n);
 printf("\n Enter Burst Time:\n");
 for(i=0;i<n;i++)
 {
  printf("p%d : ",i+1);
  scanf("%d",&bt[i]);
  }
 wt[0]=0;       //waiting time for first process will be zero
 //calculate waiting time for remaining process
 for(i=1; i<n; i++)
 {
  wt[i]=0;
  for(j=0; j<i; j++)
         wt[i]+=bt[j];
  total += wt[i];
  }
 awt = (float)total / n;          //average waiting time
 total = 0;
 printf("\n Process\t Burst Time \t Waiting Time\t Turnaround Time");
 for(i=0; i<n; i++)
 {
  tat[i] = bt[i] + wt[i];          //calculate turnaround time
  total += tat[i];
  printf("\n  p%d \t\t %d \t\t %d\t\t %d", i+1, bt[i], wt[i], tat[i]);
  }
 atat=(float)total / n;            //average turnaround time
 printf("\n\n Average Waiting Time    = %.2f",awt);
 printf("\n Average Turnaround Time = %.2f\n",atat);
 }
```

OUTPUT:

```
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ gcc fcfs.c
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ ./a.out
Enter the total number of processes: 4
Enter Burst Time:
p1 : 10
p2 : 1
p3 : 2
p4 : 1
p5 : 5
Process     Burst Time     Waiting Time  Turn Around Time
p1          10             0             10
p2          1              10            11
p3          2              11            13
p4          1              13            14
p5          5              14            19
Average Waiting Time       = 9.60
Average Turnaround Time    = 13.40
```

RESULT:

Thus the program for FCFS (First Come First Serve) CPU scheduling was implemented and verified.

1. (b) SJF (Shortest Job First) CPU Scheduling Algorithm

AIM:

Write a C program to implement the SJF CPU scheduling.

ALGORITHM:

1. Start the process.
2. Accept the number of processes in the Ready Queue
3. For each process in the Ready Queue, assign the process id and accept the CPU burst time.
4. Start the Ready Queue according the shortest Burst Time by sorting according to lowest to highest.
5. Set the waiting time of the first process as '0' and its turnaround time as its burst time.
6. For each process in the ready queue, calculate
   (a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
   (b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

7. Calculate
   (a) Average waiting time = Total waiting Time / Number of process
   (b) Average Turnaround time = Total Turnaround Time / Number of process

8. Stop the process.

PROGRAM:

```
#include<stdio.h>
void main()
{
 int bt[10], p[10], wt[10], tat[10], i, j, n, total=0, pos, temp;
 float awt,atat;
 printf(" Enter the 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;      //contains process number
  }
//sorting burst time in ascending order using selection sort
 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;     //waiting time for first process will be zero
 //calculate waiting time
 for(i=1; i<n; i++)
 {
  wt[i] = 0;
  for(j=0; j<i; j++)
          wt[i] += bt[j];
  total += wt[i];
  }
```

```
awt = (float)total / n;          //average waiting time
total = 0;
printf("\n Process \t Burst Time \t Waiting Time \t Turn Around Time");
for(i=0; i<n; i++)
{
 tat[i] = bt[i] + wt[i];          //calculate turnaround time
 total += tat[i];
 printf("\n p%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
 }
atat = (float)total / n; //average turnaround time
printf("\n\n Average Waiting Time         =%.2f",awt);
printf("\n Average Turnaround Time=%.2f\n",atat);
}
```

OUTPUT:

```
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ gcc sjf.c
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ ./a.out
Enter the total number of processes: 5
Enter Burst Time:
p1 : 10
p2 : 1
p3 : 2
p4 : 1
p5 : 5
Process        Burst Time     Waiting Time  Turn Around Time
p2             1              0                   1
p4             1              1                   2
p3             2              2                   4
p5             5              4                   9
p1             10             9                   19
Average Waiting Time          = 3.20
Average Turnaround Time       = 7.00
```

RESULT:

Thus the program for SJF (Shortest Job First) CPU scheduling was implemented and verified.

1. (c) Round Robin CPU Scheduling Algorithm

AIM:

Write a C program to implement the Round Robin CPU scheduling.

ALGORITHM:

1. Start the process.
2. Accept the number of processes in the Ready Queue and time quantum (or) time slice
3. For each process in the Ready Queue, assign the process id and accept the CPU burst time
4. Calculate the no. of time slices for each process where
5. No. of time slice for process(n) = burst time process(n) / time slice
6. If the burst time is less than the time slice then the no. of time slices =1.
7. Consider the Ready Queue is a Circular Queue, 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) Turn around time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

8. Calculate
   (a) Average waiting time = Total waiting Time / Number of process
   (b) Average Turnaround time = Total Turnaround Time / Number of process
9. Stop the process.

PROGRAM:

```
#include<stdio.h>
void main()
{
 int st[10], bt[10], wt[10], tat[10], n, tq, i, count=0, swt=0, stat=0, temp, sq=0;
 float awt, atat;
 printf("Enter the number of processes: ");
 scanf("%d",&n);
 printf("Enter the burst time of each process: \n");
 for(i=0; i<n; i++)
 {
  printf("p%d : ",i+1);
  scanf("%d",&bt[i]);
  st[i] = bt[i];
  }
 printf("Enter the time quantum: ");
 scanf("%d",&tq);
 while(1)
 {
 for(i=0,count=0; i<n; i++)
 {
  temp = tq;
  if(st[i] == 0)
  {
   count++;
   continue;
   }
  if(st[i] > tq)
     st[i] = st[i] - tq;
  else if(st[i] >= 0)
  {
   temp = st[i];
   st[i] = 0;
   }
  sq = sq + temp;
  tat[i] = sq;
  }
  if(n == count)
     break;
  }
```

```
for(i=0; i<n; i++)
{
 wt[i] = tat[i] - bt[i];
 swt = swt + wt[i];
 stat = stat + tat[i];
 }
awt = (float)swt / n;
atat = (float)stat / n;
printf("Process \t Burst Time \t Waiting Time \t Turn Around Time \n");
for(i=0; i<n; i++)
    printf(" %d \t\t %d \t\t %d \t\t %d \n",i+1,bt[i],wt[i],tat[i]);
printf("\n Average Waiting Time     = %.2f",awt);
printf("\n Average Turn Around Time = %.2f",atat);
}
```

OUTPUT:

```
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ gcc rr.c
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ ./a.out
Enter the total number of processes: 5
Enter Burst Time:
p1 : 10
p2 : 1
p3 : 2
p4 : 1
p5 : 5
Process         Burst Time      Waiting Time  Turn Around Time
p1              10              9                     19
p2              1               1                     2
p3              2               5                     7
p4              1               3                     4
p5              5               9                     14
Average Waiting Time            = 5.40
Average Turnaround Time         = 9.20
```

RESULT:

Thus the program for Round Robin CPU scheduling was implemented and verified.

1. (d) Priority CPU Scheduling Algorithm

AIM:

Write a C program to implement the Priority CPU scheduling.

ALGORITHM:

1. Start the process.
2. Accept the number of processes in the Ready Queue.
3. For each process in the Ready Queue, assign the process id and accept the CPU burst time with priority order.
4. Sort the ready queue according to the priority number.
5. Set the waiting of the first process as '0' and its burst time as its turn around time
6. For each process in the Ready Queue, calculate
   (a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)
   (b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
7. Calculate
   (a) Average waiting time = Total waiting Time / Number of process
   (b) Average Turnaround time = Total Turnaround Time / Number of process
8. Stop the process

PROGRAM:

```
#include<stdio.h>
void main()
{
 int bt[10], p[10], wt[10], tat[10], pri[10], i, j, k, n, total=0, pos, temp;
 float awt,atat;
 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;              //contains process number
  }
 printf(" Enter priority of the process:\n");
 for(i=0; i<n; i++)
 {
  p[i] = i+1;
  printf("p%d : ",i+1);
  scanf("%d",&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;
   }
 wt[0] = 0;               //waiting time for first process will be zero
 for(i=1; i<n; i++)
 {
  wt[i] = 0;
```

```
 for(j=0; j<i; j++)
          wt[i] += bt[j];
 total += wt[i];
 }
awt = (float)total / n;          //average waiting time
total = 0;
printf("\nProcess\t Burst Time \tPriority \tWaiting Time\tTurnaround Time");
for(i=0; i<n; i++)
{
 tat[i] = bt[i] + wt[i];          //calculate turnaround time
 total += tat[i];
 printf("\n p%d \t\t %d \t\t %d \t\t %d \t\t %d",p[i],bt[i],pri[i],wt[i],tat[i]);
 }
atat = (float)total / n; //average turnaround time
printf("\n\n Average Waiting Time   =%.2f",awt);
printf("\n Average Turnaround Time=%.2f\n",atat);
}
```

OUTPUT:

```
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ gcc priority.c
mrce@mrce-ThinkCentre-neo-50s-Gen-3:-/cd$ ./a.out
Enter the total number of processes: 5
Enter Burst Time:
p1 : 10
p2 : 1
p3 : 2
p4 : 1
p5 : 5
Enter priority of the process:
p1 : 3
p2 : 1
p3 : 3
p4 : 4
p5 : 2
Process     Burst Time    Priority    Waiting Time    Turn Around Time
P2          1             1           0               1
P5          5             2           1               6
p3          2             3           6               8
p1          10            3           8               18
p4          1             4           18              19
Average Waiting Time        = 6.60
Average Turnaround Time     = 10.40
```

RESULT:

Thus the program for Priority CPU scheduling was implemented and verified.

2. (a) OPEN, READ, WRITE, CLOSE - I/O SYSTEM CALLS

AIM:

C program using open, read, write, close system calls

DESCRIPTION:

1.Create:

Used to Create a new empty file

Syntax :

int creat(char *filename, mode_t mode)

filename : name of the file which you want to create

mode : indicates permissions of new file.

2.open:

Used to Open the file for reading, writing or both.

Syntax:

int open(char *path, int flags [, int mode]);

Path : path to file which you want to use

flags : How you like to use

O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write,
O_CREAT: create file if it doesn't exist, O_EXCL: prevent creation if it already exists

3. close:

Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax:

int close(int fd);

fd :file descriptor

4.read:

From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax:

int read(int fd, char *buf, int size);

fd: file descripter

buf: buffer to read data from

cnt: length of buffer

5. write:

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

Syntax:

int write(int fd, char *buf, int size);

fd: file descripter

buf: buffer to 0write data to

cnt: length of buffer

***File descriptor** is integer that uniquely identifies an open file of the process.

ALGORITHM:

1. Star the program.
2. Open a file for O_RDWR for R/W,O_CREATE for creating a file, O_TRUNC for truncate a file.
3. Using getchar(), read the character and stored in the string[] array.
4. The string [] array is write into a file close it.
5. Then the first is opened for read only mode and read the characters and displayed it and close the file.
6. Stop the program.

PROGRAM:

```
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
int main()
{
 int n,i=0;
 int f1,f2;
 char c,strin[100];
 f1=open("data",O_RDWR|O_CREAT|O_TRUNC);
 while((c=getchar())!='\n')
          strin[i++]=c;
 strin[i]='\0';
 write(f1,strin,i);
 close(f1);
 f2=open("data",O_RDONLY);
 read(f2,strin,0);
 printf("\n%s\n",strin);
 close(f2);
 return 0;
 }
```

OUTPUT:

Hai
Hai

RESULT:

Thus the program for system calls (such as open, read, write, close) was implemented and verified.

2. (b) LSEEK - I/O SYSTEM CALLS

AIM:

C program using lseek

DESCRIPTION:

lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

Syntax :

off_t lseek(int fildes, off_t offset, int whence);

int fildes : The file descriptor of the pointer that is going to be moved.

off_t offset : The offset of the pointer (measured in bytes).

int whence : Legal values for this variable are provided at the end which are SEEK_SET (Offset is to be measured in absolute terms), SEEK_CUR (Offset is to be measured relative to the current location of the pointer), SEEK_END (Offset is to be measured relative to the end of the file)

ALGORITHM:

1. Start the program
2. Open a file in read mode
3. Read the contents of the file
4. Use lseek to change the position of pointer in the read process
5. Stop

PROGRAM:

```
#include<stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
int main()
{
 int file=0;
 if((file=open("testfile.txt",O_RDONLY)) < -1)
          return 1;
 char buffer[19];
 if(read(file,buffer,19) != 19)
          return 1;
 printf("%s\n",buffer);
 if(lseek(file,10,SEEK_SET) < 0)
          return 1;
 if(read(file,buffer,19) != 19)
          return 1;
 printf("%s\n",buffer);
 return 0;
}
```

OUTPUT:

```
[188r1a0501@localhost ~]$ vi testfile.txt
[188r1a0501@localhost ~]$ cat testfile.txt
sdfghjkkl;mnbbvrtyuijnnb
ggtyujjg

[188r1a0501@localhost ~]$ gcc  w2c.c
[188r1a0501@localhost ~]$ ./a.out
sdfghjkkl;mnbbvrtyu
mnbbvrtyuijnnb
ggty
```

RESULT:

Thus the program for lseek- system call was implemented and verified.

2. (c) OPENDIR(), CLOSEDIR(), READDIR() - I/O SYSTEM CALLS

AIM:

C program using opendir(), closedir(), readdir()

DESCRIPTION:

1. Creating directories.

   Syntax : int mkdir(const char *pathname, mode_t mode);

   The ‘pathname’ argument is used for the name of the directory.

2. Opening directories

   Syntax : DIR *opendir(const char *name);

3. Reading directories.

   Syntax: struct dirent *readdir(DIR *dirp);

4. Removing directories.

   Syntax: int rmdir(const char *pathname);

5. Closing the directory.

   Syntax: int closedir(DIR *dirp);

6. Getting the current working directory.

   Syntax: char *getcwd(char *buf, size_t size);

ALGORITHM:

1. Start the program
2. Print a menu to choose the different directory operations
3. To create and remove a directory ask the user for name and create and remove the same respectively.
4. To open a directory check whether directory exists or not. If yes open the directory .If it does not exists print an error message.
5. Finally close the opened directory.
6. Stop

PROGRAM:

```
#include<stdio.h>
#include<fcntl.h>
#include<dirent.h>
main()
{
char d[10]; int c,op; DIR *e;
struct dirent *sd;
printf("**menu**\n1.create dir\n2.remove dir\n 3.read dir\n enter ur choice");
scanf("%d",&op);
switch(op)
{
 case 1:
	printf("enter dir name\n"); scanf("%s",&d);
	c=mkdir(d,777);
	if(c==1)
		printf("dir is not created");
	else
		printf("dir is created"); break;
 case 2:
	printf("enter dir name\n"); scanf("%s",&d);
	c=rmdir(d);
	if(c==1)
		printf("dir is not removed");
	else
		printf("dir is removed"); break;
 case 3:
	printf("enter dir name to open");
	scanf("%s",&d);
	e=opendir(d);
	if(e==NULL)
		printf("dir does not exist");
	else
	{
	 printf("dir exist\n");
	 while((sd=readdir(e))!=NULL)
		printf("%s\t",sd->d_name);
	 }
	closedir(e);
	break;
 }
}
```

OUTPUT:

```
[188r1a0501@localhost f]$ gcc  w2e.c
[188r1a0501@localhost f]$ ./a.out
**menu**
1.create dir
2.remove dir
 3.read dir
 enter ur choice1
enter dir name
d
dir is created[188r1a0501@localhost f]$ ls
a.out  a.txt  d  w2d.c  w2e.c
```

RESULT:

Thus the program for opendir(), closedir(), readdir() - system calls was implemented and verified.

3. BANKERS ALGORITHM FOR DEADLOCK AVOIDANCE AND PREVENTION

AIM:

Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.

ALGORITHM:

1. Start the program.
2. Get or assign the values of resources allocated, max and available.
3. Find the need value.
4. Check whether it is possible to allocate.
5. If it is possible then the system is in safe state.
6. Else system is not in safety state.
7. If the new request comes then check that the system is in safety or not.
8. Stop the program.

PROGRAM:

```
#include <stdio.h>
#define n 5   // No.of processes
#define m 3    // No.of resouces
int main()
{
  // P0, P1, P2, P3, P4 are the Process names here
    int i, j, k, y=0, need[n][m], flag=1, f[n], ans[n], ind=0;
    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

    for (k = 0; k < n; k++)
          f[k] = 0;

    for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
             need[i][j] = max[i][j] - alloc[i][j];

    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;
        }
    }
   }
  }

for(i=0;i<n;i++)
{
 if(f[i]==0)
 {
  flag=0;
  printf("The system is UNSAFE state");
  break;
  }
 }

if(flag==1)
{
 printf("The system is SAFE and safe sequence is \t");
  for (i = 0; i < n - 1; i++)
      printf(" P%d ->", ans[i]);
  printf(" P%d", ans[n - 1]);
 }
return (0);
}
```

OUTPUT:

The system is SAFE and safe sequence is P1 -> P3 -> P4 -> P0 -> P2

RESULT:

Thus, the program of Bankers Algorithm for Deadlock Avoidance was executed and verified.

4. PRODUCER – CONSUMER PROBLEM USING SEMAPHORES

AIM:

Write a C program to implement the Producer – Consumer problem (using semaphores) using UNIX/LINUX system calls.

ALGORITHM:

1. The Semaphore mutex, full & empty are initialized.
2. In the case of producer process
3. Produce an item in to temporary variable.
   i. If there is empty space in the buffer check the mutex value for enter into the critical section.
   ii. If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.
4. In the case of consumer process
   i) It should wait if the buffer is empty
   ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
   iii) Signal the mutex value and reduce the empty value by 1.
   iv) Consume the item.
5. Print the result

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 3, x = 0;

int main ()
{
 int n;
 void producer ();
 void consumer ();
 int wait (int);
 int signal (int);
 printf ("\n 1.Producer \n 2.Consumer \n 3.Exit");
 while (1)
 {
  printf ("\n\t Enter your choice: ");
  scanf ("%d", &n);
  switch (n)
  {
   case 1:
        if ((mutex == 1) && (empty != 0))
                producer ();
        else
                printf ("\t\t Buffer is FULL !!!");
        break;
   case 2:
        if ((mutex == 1) && (full != 0))
                consumer ();
        else
                printf ("\t\t Buffer is EMPTY !!!");
        break;
   case 3:
        exit (0);
   }
  }
 return 0;
 }

int wait (int s)
{
 return (--s);
 }
```

```
int signal (int s)
{
 return (++s);
 }

void producer ()
{
 mutex = wait (mutex);
 full = signal (full);
 empty = wait (empty);
 x++;
 printf ("\n Producer produces the item %d", x);
 mutex = signal (mutex);
 }

void consumer ()
{
 mutex = wait (mutex);
 full = wait (full);
 empty = signal (empty);
 printf ("\n Consumer consumes item %d", x);
 x--;
 mutex = signal (mutex);
 }
```

OUTPUT:

```
[188r1a0501@localhost ~]$ vi pc.c
[188r1a0501@localhost ~]$ gcc  pc.c
[188r1a0501@localhost ~]$ ./a.out

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the 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
[188r1a0501@localhost ~]$
```

RESULT:

Thus, the program for Producer – Consumer problem using semaphores was executed and verified.

5. IPC MECHANISMS

AIM:

Write C programs to illustrate the following IPC mechanisms

a) Pipes   b) FIFOs   c) Message Queues   d) Shared Memory

ALGORITHM:

1. Start the program.
2. Initialize IPC mechanism (create necessary data structures, semaphores, etc.).
3. Process A (Sender):
   a. Prepare the data/message to be sent.
   b. Acquire a lock or semaphore to access the IPC channel.
   c. Write the data/message to the shared IPC buffer or channel.
   d. Release the lock or semaphore.
   e. Signal or notify Process B about the availability of new data.
4. Process B (Receiver):
   a. Wait for a signal or notification from Process A.
   b. Acquire a lock or semaphore to access the IPC channel.
   c. Read the data/message from the shared IPC buffer or channel.
   d. Process or use the received data/message.
   e. Release the lock or semaphore.
5. Clean up and terminate IPC mechanism.
6. Stop.

(a) Program – IPC Mechanism using PIPE

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

int main()
{
   int fd[2];
   char buffer[50];

   if (pipe(fd) == -1)
   {
      perror("pipe");
      return 1;
   }

   if (fork() == 0)              // Child process
   {
      close(fd[0]);              // Close read end
      char msg[] = "Hello from child process!";
      write(fd[1], msg, sizeof(msg));
      close(fd[1]);
   }
   else                          // Parent process
   {
      close(fd[1]);              // Close write end
      read(fd[0], buffer, sizeof(buffer));
      printf("Parent received: %s\n", buffer);
      close(fd[0]);
   }

   return 0;
}
```

OUTPUT:

```
Parent received: Hello from child process!
```

(b) Program – IPC Mechanism using FIFO

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
   const char *fifoFile = "/tmp/myfifo";
   char buffer[50];

   mkfifo(fifoFile, 0666);

   int fd = open(fifoFile, O_RDWR);

   if (fork() == 0)                    // Child process
   {
      char msg[] = "Hello from child process!";
      write(fd, msg, sizeof(msg));
      close(fd);
    }
   else                                // Parent process
  {
      read(fd, buffer, sizeof(buffer));
      printf("Parent received: %s\n", buffer);
      close(fd);
   }
   return 0;
 }
```

OUTPUT:

```
Parent received: Hello from child process!
```

(c) Program – IPC Mechanism using MESSAGE QUEUE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message
{
   long mtype;
   char mtext[50];
};
int main()
{
   key_t key;
   int msgid;
   struct message msg;

   key = ftok("msgq_file", 'b');
   msgid = msgget(key, 0666 | IPC_CREAT);
   if (fork() == 0)                    // Child process
   {
      struct message childMsg;
      childMsg.mtype = 1;
      strcpy(childMsg.mtext, "Hello from child process!");
      msgsnd(msgid, &childMsg, sizeof(childMsg.mtext), 0);
   }
   else                                // Parent process
   {
      msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0);
      printf("Parent received: %s\n", msg.mtext);
      msgctl(msgid, IPC_RMID, NULL);
   }
   return 0;
}
```

OUTPUT:

```
Parent received: Hello from child process!
```

(d) Program – IPC Mechanism using SHARED MEMORY

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    key_t key;
    int shmid;
    char *shmaddr;
    key = ftok("shm_file", 'b');
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    shmaddr = shmat(shmid, (void *)0, 0);
    if (fork() == 0)                         // Child process
   {
        strcpy(shmaddr, "Hello from child process!");
        shmdt(shmaddr);
    }
   else                                      // Parent process
    {
        wait(NULL);
        printf("Parent received: %s\n", shmaddr);
        shmdt(shmaddr);
        shmctl(shmid, IPC_RMID, NULL);
    }
    return 0;
}
```

OUTPUT:

Parent received: Hello from child process!

RESULT:

Thus the programs for IPC mechanisms (Pipes, FIFOs, Message Queues, Shared Memory) was executed and verified.

6. a) PAGING – MEMORY MANAGEMENT TECHNIQUE

AIM:

To write a C program to implement memory management using paging technique.

ALGORITHM:

1. Start the program.
2. Read/Assign the base address, page size, number of pages and memory unit.
3. If the memory limit is less than the base address display the memory limit is less than limit.
4. Create the page table with the number of pages and page address.
5. Read the page number and displacement value.
6. If the page number and displacement value is valid, add the displacement value with the address corresponding to the page number and display the result.
7. Display the page is not found or displacement should be less than page size.
8. Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_FRAMES 4
#define PAGE_SIZE  256
#define MEMORY_SIZE NUM_FRAMES * PAGE_SIZE
typedef struct
{
   int valid;
   int frame;
 } PageTableEntry;

int main()
{
   PageTableEntry pageTable[16];
   char memory[MEMORY_SIZE];
   for (int i = 0; i < 16; i++)                    // Initialize page table
  {
      pageTable[i].valid = 0;
      pageTable[i].frame = -1;
   }

   for (int i = 0; i < 16; i++)                    // Simulate memory allocation
  {
      int pageNumber = i;
      int pageIndex = pageNumber * PAGE_SIZE;
      if (!pageTable[pageNumber].valid)
     {
        for (int j = 0; j < NUM_FRAMES; j++)             // Page fault
        {
           if (pageTable[j].frame == -1)
          {
              pageTable[j].frame = pageIndex;
              pageTable[j].valid = 1;
              printf("Page %d loaded into Frame %d\n", pageNumber, j);
              break;
           }
        }
     }
```

```
        // Access memory using the page table
        char value=memory[pageTable[pageNumber].frame+ (i % PAGE_SIZE)];
        printf("Accessing Memory at Page %d, Offset %d: Value = %d\n",
                                    pageNumber, i % PAGE_SIZE, value);
    }
    return 0;
}
```

OUTPUT:

```
Page 0 loaded into Frame 0
Accessing Memory at Page 0, Offset 0: Value = 48
Page 1 loaded into Frame 1
Accessing Memory at Page 1, Offset 1: Value = -125
Page 2 loaded into Frame 2
Accessing Memory at Page 2, Offset 2: Value = 0
Page 3 loaded into Frame 3
Accessing Memory at Page 3, Offset 3: Value = 0
Accessing Memory at Page 4, Offset 4: Value = -124
Accessing Memory at Page 5, Offset 5: Value = -36
Accessing Memory at Page 6, Offset 6: Value = 127
Accessing Memory at Page 7, Offset 7: Value = 0
Accessing Memory at Page 8, Offset 8: Value = 0
Accessing Memory at Page 9, Offset 9: Value = -104
Accessing Memory at Page 10, Offset 10: Value = -64
Accessing Memory at Page 11, Offset 11: Value = -14
Accessing Memory at Page 12, Offset 12: Value = -124
Accessing Memory at Page 13, Offset 13: Value = -36
Accessing Memory at Page 14, Offset 14: Value = 127
Accessing Memory at Page 15, Offset 15: Value = 0
```

RESULT:

Thus the program for memory management using paging technique was executed and verified.

6. b) SEGMENTATION – MEMORY MANAGEMENT TECHNIQUE

AIM:

To write a C program to implement memory management using segmentation technique.

ALGORITHM:

1. Start the program.
2. Read/Assign the base address, number of segments, size of each segment, memory limit.
3. If memory address is less than the base address display “invalid memory limit”.
4. Create the segment table with the segment number and segment address and display it.
5. Read the segment number and displacement.
6. If the segment number and displacement is valid compute the real address and display the same.
7. Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_SEGMENTS 4
#define SEGMENT_SIZE 1024
#define MEMORY_SIZE NUM_SEGMENTS * SEGMENT_SIZE
typedef struct
{
   int base;
   int limit;
} SegmentDescriptor;

int main()
{
   SegmentDescriptor segmentTable[4];
   char memory[MEMORY_SIZE];
  for (int i = 0; i < 4; i++)                    // Initialize segment table
  {
      segmentTable[i].base = -1;
      segmentTable[i].limit = -1;
   }
   for (int i = 0; i < 4; i++)                      // Simulate memory allocation
  {
      int segmentNumber = i;
      if (segmentTable[segmentNumber].base == -1)
      {
         segmentTable[segmentNumber].base = i * SEGMENT_SIZE;
         segmentTable[segmentNumber].limit = SEGMENT_SIZE;
         printf("Segment %d created at Base Address %d\n", segmentNumber,
                        segmentTable[segmentNumber].base);
      }

      // Access memory using the segment table
      int offset = i * 256;  // Each segment has 256 bytes
      char value = memory[segmentTable[segmentNumber].base + offset];
      printf("Accessing Memory at Segment %d, Offset %d: Value = %d\n",
                              segmentNumber, offset, value);
   }
   return 0;
}
```

OUTPUT:

Segment 0 created at Base Address 0
Accessing Memory at Segment 0, Offset 0: Value = 0
Segment 1 created at Base Address 1024
Accessing Memory at Segment 1, Offset 256: Value = -128
Segment 2 created at Base Address 2048
Accessing Memory at Segment 2, Offset 512: Value = 0
Segment 3 created at Base Address 3072
Accessing Memory at Segment 3, Offset 768: Value = 0

RESULT:

Thus the program for memory management using segmentation technique was executed and verified.