Aim:Write a C program to implement the various process scheduling mechanisms such as FCFS,SJF,Priority .

Algorithm for FCFS scheduling:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

Average waiting time = Total waiting Time / Number of process

Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
int nop,wt[10],twt,tat[10],ttat,i,j,bt[10],t;
float awt,atat;
clrscr();
awt=0.0;
atat=0.0;
printf("Enter the no.of process:");
scanf("%d",&nop);
for(i=0;i<nop;i++)
{
printf("Enter the burst time for process %d: ", i);
scanf("%d",&bt[i]);
}
wt[0]=0;
tat[0]=bt[0];
twt=wt[0];
ttat=tat[0];
for(i=1;i<nop;i++){
wt[i]=wt[i-1]+bt[i-1];
tat[i]=wt[i]+bt[i];
twt+=wt[i];
ttat+=tat[i];}
awt=(float)twt/nop;
atat=(float)ttat/nop;
printf("\nProcessid\tBurstTime\tWaitingTime\tTurnaroundTime\n");
for(i=0;i<nop;i++)
printf("%d\t\t%d\t\t%d\t\t%d\n",i,bt[i],wt[i],tat[i]);
printf("\nTotal Waiting Time:%d\n",twt);
printf("\nTotal Around Time:%d\n",ttat);
printf("\nAverage Waiting Time:%f\n",awt);
printf("\nAverage Total Around Time:%f\n",atat);
getch();}
```

Algorithm for SJF

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate

Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate Average waiting time = Total waiting Time / Number of process

Average Turnaround time = Total Turnaround Time / Number of process

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main(){
int nop,wt[10],twt,tat[10],ttat,i,j,bt[10],t;
float awt,atat;
clrscr();
awt=0.0;
atat=0.0;
printf("Enter the no.of process:");
scanf("%d",&nop);
for(i=0;i<nop;i++){
printf("Enter the burst time for process %d: ", i);
scanf("%d",&bt[i]);}
for(i=0;i<nop;i++){
for(j=i+1;j<nop;j++){
if(bt[i]>=bt[j]){
t=bt[i];
bt[i]=bt[j];
bt[j]=t;}}}
wt[0]=0;
tat[0]=bt[0];
twt=wt[0];
ttat=tat[0];
for(i=1;i<nop;i++){
wt[i]=wt[i-1]+bt[i-1];
tat[i]=wt[i]+bt[i];
twt+=wt[i];
ttat+=tat[i];}
awt=(float)twt/nop;
atat=(float)ttat/nop;
printf("\nProcessid\tBurstTime\tWaitingTime\tTurnaroundTime\n");
for(i=0;i<nop;i++)
printf("%d\t\t%d\t\t%d\t\t%d\n",i,bt[i],wt[i],tat[i]);
printf("\nTotal Waiting Time:%d\n",twt);
printf("\nTotal Around Time:%d\n",ttat);
printf("\nAverage Waiting Time:%f\n",awt);
printf("\nAverage Total Around Time:%f\n",atat);
getch();}
```

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

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(){
char s[21][21],chng[20];
int wt[21],a[21],n,i,j,temp,trn[21],p[21];
float tot,t;
printf("Enter the no.of process");
scanf("%d",&n);
for(i=1;i<=n;i++){
printf("Enter process id and time and priority");
scanf("%s%d%d",&s[i],&a[i],&p[i]);}
wt[0]=0;
a[0]=0;
t=tot=0;
for(i=1;i<=n;i++){
for(j=i+1;j<=n;j++){
if(p[i]>p[j]){
temp=a[i];
a[i]=a[j];
a[j]=temp;
temp=p[i];
p[i]=p[j];
p[j]=temp;
strcpy(chng,s[i]);
strcpy(s[i],s[j]);
strcpy(s[j],chng);}}}
printf(" \n process\t burst time\t waiting time\t turn around time\t priority");
for(i=1;i<=n;i++){
wt[i]=wt[i-1]+a[i-1];
trn[i]=wt[i]+a[i];
printf("%s\t%d\t%d\t%d\t%d\n",s[i],a[i],wt[i],trn[i],p[i]);
tot=tot+wt[i];
t=t+trn[i];}
printf("Average waiting time=%f Average turn around time=%f",(tot/n),(t/n));
getch();
}
```

Aim: Write a program to solve the Dining Philosophers problem.

Algorithm: 1.Initialize the state array S as 0, S_i =0 if the philosopher i is thinking or 1 if hungry.
2.Associate two functions getfork(i) and putfork(i) for each philosopher i.
3.For each philiosopher I call getfork(i) , test(i) and putfork(i) if i is 0. 4.Stop
Algorithm for getfork(i): Step 1: set S[i]= 1 i.e. the philosopher i is hungry
Step 2: call test(i)
Algorithm for putfork(i): Step 1: set S[i]=0 I.e. the philosopher i is thinking
Step 2: test(LEFT) and test(RIGHT)
Algorithm for test(i) : Step 1: check if (state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
Step 2: give the i philosopher a chance to eat.

PROBLEM

```
#include<stdio.h>
#include<conio.h>
int LEFT;
int RIGHT;
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[5];
void put_forks(int);
void test(int);
void take_forks(int);
void philosopher(int i){
if(state[i]==0){
take_forks(i);
if(state[i]==EATING)
printf("\n Eating in process... ");
put_forks(i); } }
void put_forks(int i){
state[i]=THINKING;
printf("\n philosopher %d completed its works",i);
test(LEFT);
test(RIGHT);}
void take_forks(int i){
state[i]=HUNGRY;
test(i);}
void test(int i){
if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING){
printf("\n philosopher %d can eat",i);
state[i]=EATING;}}
void main(){
int i;
for(i=1;i<=5;i++)
state[i]=0;
printf("\n\t\t\t Dining Philosopher Problem");
printf("\n\t\t. ..... ");
for(i=1;i<=5;i++){
printf("\n\n the philosopher %d falls hungry\n",i);
philosopher(i);}
getch();}
```

AIM: To implement memory allocation techniques using
1.First fit 2.Best fit 3.Worst fit & 4.To make comparative study
ALGORITHM:
Step 2: Get the number of memory partition and their sizes.
Step 3: Get the number of processes and values of block size for each process.
Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.
Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.
Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
Step 8: Stop the program.

PROGRAM

First fit

```
#include<stdio.h>

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
   int i, j;
   // Stores block id of the
   // block allocated to a process
   int allocation[n];

   // Initially no block is assigned to any process
   for(i = 0; i < n; i++)
   {
      allocation[i] = -1;
   }

   // pick each process and find suitable blocks
   // according to its size ad assign to it
   for (i = 0; i < n; i++)       //here, n -> number of processes
   {
      for (j = 0; j < m; j++)       //here, m -> number of blocks
      {
         if (blockSize[j] >= processSize[i])
         {
            // allocating block j to the ith process
            allocation[i] = j;

            // Reduce available memory in this block.
            blockSize[j] -= processSize[i];

            break;    //go to the next process in the queue
         }
      }
   }

   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < n; i++)
   {
      printf(" %i\t\t\t", i+1);
      printf("%i\t\t\t\t", processSize[i]);
      if (allocation[i] != -1)
         printf("%i", allocation[i] + 1);
      else
```

```
            printf("Not Allocated");
        printf("\n");
    }
}

// Driver code
int main()
{
    int m;    //number of blocks in the memory
     int n;  //number of processes in the input queue
     int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    m = sizeof(blockSize) / sizeof(blockSize[0]);
    n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

Best fit

```
#include <stdio.h>

void implimentBestFit(int blockSize[], int blocks, int processSize[], int proccesses)
{
    // This will store the block id of the allocated block to a process
    int allocation[proccesses];
    int occupied[blocks];

    // initially assigning -1 to all allocation indexes
    // means nothing is allocated currently
    for(int i = 0; i < proccesses; i++){
        allocation[i] = -1;
    }

    for(int i = 0; i < blocks; i++){
        occupied[i] = 0;
    }

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < proccesses; i++)
    {

        int indexPlaced = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i] && !occupied[j])
            {
                // place it at the first block fit to accomodate process
                if (indexPlaced == -1)
                    indexPlaced = j;

                // if any future block is smalller than the current block where
                // process is placed, change the block and thus indexPlaced
                            // this reduces the wastage thus best fit
                else if (blockSize[j] < blockSize[indexPlaced])
                    indexPlaced = j;
            }
        }

        // If we were successfully able to find block for the process
        if (indexPlaced != -1)
```

```
      {
         // allocate this block j to process p[i]
         allocation[i] = indexPlaced;

         // make the status of the block as occupied
         occupied[indexPlaced] = 1;
      }
   }

   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < proccesses; i++)
   {
      printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
      if (allocation[i] != -1)
         printf("%d\n",allocation[i] + 1);
      else
         printf("Not Allocated\n");
   }
}

// Driver code
int main()
{
   int blockSize[] = {100, 50, 30, 120, 35};
   int processSize[] = {40, 10, 30, 60};
   int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
   int proccesses = sizeof(processSize)/sizeof(processSize[0]);

   implimentBestFit(blockSize, blocks, processSize, proccesses);

   return 0 ;
}
```

Worst fit

```
#include <stdio.h>

void implimentWorstFit(int blockSize[], int blocks, int processSize[], int processes)
{
   // This will store the block id of the allocated block to a process
   int allocation[processes];
   int occupied[blocks];

   // initially assigning -1 to all allocation indexes
   // means nothing is allocated currently
   for(int i = 0; i < processes; i++){
      allocation[i] = -1;
   }

   for(int i = 0; i < blocks; i++){
      occupied[i] = 0;
   }

   // pick each process and find suitable blocks
   // according to its size ad assign to it
   for (int i=0; i < processes; i++)
   {
            int indexPlaced = -1;
            for(int j = 0; j < blocks; j++)
            {
               // if not occupied and block size is large enough
               if(blockSize[j] >= processSize[i] && !occupied[j])
```

```
            {
                // place it at the first block fit to accomodate process
                if (indexPlaced == -1)
                    indexPlaced = j;

                // if any future block is larger than the current block where
                // process is placed, change the block and thus indexPlaced
                else if (blockSize[indexPlaced] < blockSize[j])
                    indexPlaced = j;
            }
        }

        // If we were successfully able to find block for the process
        if (indexPlaced != -1)
        {
            // allocate this block j to process p[i]
            allocation[i] = indexPlaced;

            // make the status of the block as occupied
            occupied[indexPlaced] = 1;

            // Reduce available memory for the block
            blockSize[indexPlaced] -= processSize[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < processes; i++)
    {
        printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n",allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

// Driver code
int main()
{
    int blockSize[] = {100, 50, 30, 120, 35};
    int processSize[] = {40, 10, 30, 60};
    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
    int processes = sizeof(processSize)/sizeof(processSize[0]);

    implimentWorstFit(blockSize, blocks, processSize, processes);

    return 0
```

Aim: To write a program to implement producer consumer problem.

Algorithm:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any cosumer.If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

PROBLEM

```
#include<stdio.h>
#include<conio.h>
int main(){
int s,n,b=0,p=0,c=0;
printf("\n producer and consumer problem");
do{
printf("\n menu");
printf("\n 1.producer an item");
printf("\n 2.consumer an item");
printf("\n 3.add item to the buffer");
printf("\n 4.display status");
printf("\n 5.exit");
printf("\n enter the choice");
scanf("%d",&s);
switch(s){
case 1:
p=p+1;
printf("\n item to be produced");
break;
case 2:
if(b!=0){
c=c+1;
b=b-1;
printf("\n item to be consumed");}
else{
printf("\n the buffer is empty please wait...");}
break;
case 3:
if(b<n){
if(p!=0){
b=b+1;
printf("\n item added to buffer");}
else
printf("\n no.of items to add...");}
else
printf("\n buffer is full,please wait");
break;
```

```
case 4:
printf("no.of items produced :%d",p);
printf("\n no.of consumed items:%d",c);
printf("\n no.of buffered item:%d",b);
break;
case 5:exit(0);}}
while(s<=5);
getch();
return 0;
}
```

Aim: To implement the Memory management policy- Paging.

Algorithm:

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following

Physical address = (Frame number * Frame size) + offset

Step 5: Display the physical address.

Step 6: Stop the process.

PROGRAM:

```
#include  <stdio.h>
#include <conio.h>
#include  <math.h>
void main()
{
   int size,m,n,pgno,pagetable[3]={5,6,7},i,j,frameno;
   double m1;
   int ra=0,ofs;
   printf("enter process size:");
   scanf("%d",&size);
   m1=size/4;
   n=ceil(m1);
   printf("total no of pages %d",n);
   printf("\n enter relative address \n:");
   scanf("%d",&ra);
   pgno=ra/1000;
   ofs=ra%1000;
   printf("pageno = %d\n", pgno);
   printf("page table");
   for(i=0;i<n;i++)
   printf("\n %d [%d]",i,pagetable[i]);
   frameno = pagetable[pgno];
   printf("\n equivalent physical address:%d%d",frameno,ofs);
   getch();

}
```

PROGRAM: calculator

```
echo “ Menu Based Calculator”
echo "Enter the Operands"
read a
read b
echo "Enter the Operator"
read o
case $o in
"+" ) echo “$a + $b” = `expr $a + $b`;;
"-" ) echo “$a - $b” = `expr $a - $b`;;
"*" ) echo “$a * $b” = `expr $a * $b`;;
"/" ) echo  “$a / $b” =  `expr $a / $b`;;
* ) echo " Inavlid Operation"
esac
```

PROGRAM system information

```
echo "SYSTEM INFORMATION"
echo “Hello ,$LOGNAME”
echo “Current Date is = $(date)”
echo “User is ‘who I am’”
echo “Current Directory = $(pwd)”
echo "Network Name and Node Name = $(uname -n)"
echo "Kernal Name =$(uname -s)"
echo "Kernal Version=$(uname -v)"
echo "Kernal Release =$(uname -r)"
echo "Kernal OS =$(uname -o)"
echo “Proessor Type = $(uname -p)”
echo “Kernel Machine Information = $(uname –m)”
echo "All Information =$(uname -a)"
```

PROGRAM printing pattern

```
echo "Enter the Limit "
read n
echo "Pattern"
for (( i = 1 ; i < $n ; i++ ))
do
for (( j = 1 ; j <= i ; j++ ))
do
echo -n " $ "
done
echo " "
done
```

PROGRAM: echo –n “Enter the Filename”

```
read filename
if [ ! -f $filename ];
then
echo “Filename $filename does not exists”
exit 1
fi
tr  ‘[A-Z]’ ‘[a-z]’ <  $filename
```

PROGRAM: substring

```
echo Enter main string:
read main
l1=`echo $main | wc -c`
l1=`expr $l1 - 1`
echo Enter sub string:
read sub
l2=`echo $sub | wc -c`
l2=`expr $l2 - 1`
n=1
m=1
pos=0
while [ $n -le $l1 ]
do
a=`echo $main | cut -c $n`
b=`echo $sub | cut -c $m`
if [ $a = $b ]
then
n=`expr $n + 1`
m=`expr $m + 1`
pos=`expr $n - $l2`
r=`expr $m - 1`
if [ $r -eq $l2 ]
then
break
fi
else
pos=0
m=1
n=`expr $n + 1`
fi
done
echo Position of sub stringin main string is $pos
```

program: print lowercase to uppercase

```
for i in *
do
echo Before Converting to uppercase the filename is
echo $i
j=`echo $i | tr '[a-z]' '[A-Z]'`
echo After Converting to uppercase the filename is
echo $j
mv $i $j
done
```

PROGRAM-manipulate date time

```
echo “hello,$LOGNAME”
echo “user is , ‘who I am’”
echo “date is,’date’”
echo “current directory ,$(pwd)”
```

AIM:write a program to implement bankers algorithm
ALGORITHM:
Step1:start
Setp2:initialize count to 0
Step3:get the no of process and resources.
Step4:get the max matrix and allocation matrix
Step5:compute the resultant matrix and print the output.
Step6: stop
Program:

```
#include<stdio.h>
#include<conio.h>
void main(){
int
k=0,output[10],d=0,t=0,ins[5],i,avail[5],allocated[10][5],need[10][5],MAX[10][5],pno,P[10],j,rz,count
=0;
printf("\nenter the number of resources: ");
scanf("%d",&rz);
printf("\nenter the max instances of each resources\n");
for(i=0;i<rz;i++){
avail[i]=0;
printf("%c= ",(i+97));
scanf("%d",&ins[i]);}
printf("\nenter the number of processes: ");
scanf("%d",&pno);
printf("\nenter the allocation matrix\n");
for(i=0;i<rz;i++)
printf("%c ",(i+97));
printf("\n");
for(i=0;i<pno;i++){
P[i]=i;
printf("P[%d] ",P[i]);
for(j=0;j<rz;j++){
scanf("%d",&allocated[i][j]);
avail[j]+=allocated[i][j];} }
printf("\nenter the MAX matrix\n");
for(i=0;i<rz;i++){
printf("%c ",(i+97));
avail[i]=ins[i]-avail[i];}
printf("\n");
for(i=0;i<pno;i++){
printf("P[%d] ",i);
for(j=0;j<rz;j++)
scanf("%d",&MAX[i][j]);}
printf("\n");
A:d==-1;
for(i=0;i<pno;i++)
count=0;
t=P[i];
for(j=0;j<rz;j++){
need[t][j]=MAX[t][j]-allocated[t][j];
if(need[t][j]<=avail[j])
```

```
count++;}
if(count==rz){
output[k++]=P[i];
for(j=0;j<rz;j++)
avail[j]+=allocated[t][j];}
else{
P[++d]=P[i];}
if(d!=-1){
pno=d+1;
goto A;}
printf("\t<");
for(i=0;i<k;i++)
printf("P[%d] ",output[i]);
printf(">");
getch();}
```

Aim:

To write a program to implement readers and writers problem

Algorithm:

Step 1: Get exclusive access to rc(lock Mutex)
Step 2: Increment rc by 1
Step 3: Get the exclusive access bd(lock bd)
Step 4: Release exclusive access to rc(unlock Mutex)
Step 5: Release exclusive access to rc(unlock Mutex)
Step 6: Read the data from database
Step 7: Get the exclusive access to rc(lock mutex)
Step 8: Decrement rc by 1, if rc =0 this is the last reader.
Step 9: Release exclusive access to database(unlock mutex)
Step 10 Release exclusive access to rc(unlock mutex)

```
#include<stdio.h>
#include<conio.h>
void main(){
typedef int semaphore;
semaphore sread=0, swrite=0;
int ch,r=0;
printf("\nReader writer");
do{
printf("\nMenu");
printf("\n\t 1.Read from file");
printf("\n \t2.Write to file");
printf("\n \t 3.Exit the reader");
printf("\n \t 4.Exit the writer");
printf("\n \t 5.Exit");
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch){
case 1: if(swrite==0){
        sread=1;
        r+=1;
        printf("\nReader %d reads",r);}
        else
```

```
            {printf("\n Not possible");}
            break;
case 2: if(sread==0 && swrite==0){
            swrite=1;
            printf("\nWriter in Progress");}
        else if(swrite==1)
            {printf("\nWriter writes the files");}
            else if(sread==1)
            {printf("\nCannot write while reader reads the file");}
            else
            printf("\nCannot write file");
            break;
case 3: if(r!=0){
            printf("\n The reader %d closes the file",r);
            r-=1;}
            else if(r==0){
            printf("\n Currently no readers access the file");
            sread=0;}
            else if(r==1){
            printf("\nOnly 1 reader file");}
            else
            printf("%d reader are reading the file\n",r);
break;
case 4: if (swrite==1){
            printf("\nWriter close the file");
            swrite=0;}
            else
            printf("\nThere is no writer in the file");
            break;
case 5: exit(0);} }
while(ch<6);
getch();}
```