**EX.NO : 1) PROGRAM USING SYSTEM CALL fork()**

**AIM :** To write the program to create a Child Process using system call fork().

**ALGORITHM :**

Step 1 : Declare the variable pid.

Step 2 : Get the pid value using system call fork().

Step 3 : If pid value is less than zero then print as “Fork failed”.

Step 4 : Else if pid value is equal to zero include the new process in the system‟s file using execlp system call.

Step 5 : Else if pid is greater than zero then it is the parent

process and it waits till the child completes using the system call wait()

Step 6 : Then print “Child complete”.

**SYSTEM CALLS USED:**

**1. fork( )**

Used to create new processes. The new process consistsof a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

**Syntax : fork( )**

**2.execlp( )**

Used after the fork() system call by one of the two processes to replace the process‟ memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution.The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.

**Syntax : execlp( )**

**3. wait( )**

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

**Syntax : wait( NULL)**

**4. exit( )**

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

**Syntax: exit(0)**

**PROGRAM CODING :**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

void main( )

{

int pid;

pid=fork();

if(pid<0)

{

printf("fork failed");

exit(1);

}

else if(pid==0)

{

execlp("whoami","ls",NULL);

exit(0);

}

else

{

printf("\n Process id is -%d\n",getpid()); wait(NULL);

exit(0);

}

}

**OUTPUT:**

[cse6@localhost Pgm]$ cc prog4a.c

[cse6@localhost Pgm]$ ./a.out

**RESULT:**

Thus the program was executed and verified successfully

**EX.NO : 2) PROGRAM USING SYSTEM CALLS getpid() & getppid()**

**AIM :**To write the program to implement the system calls getpid() and getppid().

**ALGORITHM :**

Step 1 : Declare the variables pid , parent pid , child id and grand chil id.

Step 2 : Get the child id value using system call fork().

Step 3 : If child id value is less than zero then print as “error at fork() child”.

Step 4 : If child id !=0 then using getpid() system call get the process id.

Step 5 : Print “I am parent” and print the process id.

Step 6 : Get the grand child id value using system call fork().

Step 7 : If the grand child id value is less than zero then print as “error at fork() grandchild”.

Step 8 : If the grand child id !=0 then using getpid system call get the process id.

Step 9 : Assign the value of pid to my pid.a

Step 10 : Print “I am child” and print the value of my pid.

Step 11 : Get my parent pid value using system call getppid().

Step 12 : Print “My parent‟s process id” and its value.

Step 13 : Else print “I am grand child”.

Step 14 : Get the grand child‟s process id using getpid() and print it as “my process id”.

Step 15 : Get the grand child‟s parent process id using getppid() and print it as “my parent‟s process id

**SYSTEM CALLS USED : 1.getpid( )**

Each process is identified by its id value**.** This function is used to get the id value of a particular process.

**2.getppid( )**

Used to get particular process parent‟s id value.

**PROGRAM CODING:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main( )

{

int pid;

pid=frok();

if(!pid)

{

printf(“\n Child process is under execution”);

printf(“\n Process id of the child process is %d”, getpid());

printf(“\n Process id of the parent process is %d”, getppid());

}

else

{

printf(“\n Parent process is under execution”);

printf(“\n Process id of the parent process is %d”, getpid());

printf(“\n Process id of the child process in parent is %d”, pid);

printf(“\n Process id of the parent of parent is %d”, getppid());

}

return(0);

}

**OUTPUT:**

Child process is under execution Process id of the child process is 9314

Process id of the parent process is 9313 Parent process is under execution

Process id of the parent process is 9313

Process id of the child process in parent is 9314 Process id of the parent of parent is 2825

**RESULT:**

Thus the program was executed and verified successfully

**EX.NO : 3)PROGRAM USING SYSTEM CALLS opendir( ) readdir( ) closedir()**

**AIM :**To write the program to implement the system calls opendir( ), readdir( ).

**ALGORITHM :**

Step 1 : Start.

Step 2 : In the main function pass the arguments.

Step 3 : Create structure as stat buff and the variables as integer.

Step 4 : Using the for loop,initialization

**SYSTEM CALLS USED:**

**1.opendir( )**:-Open a directory.

**2.readdir( )**:-Read a directory.

**3.closedir()**:-Close a directory.

**PROGRAM CODING:**

#include<stdio.h>

#include<sys/types.h>

#include<sys/dir.h>

void main(int age,char \*argv[])

{

DIR \*dir;

struct dirent \*rddir;

printf("\n Listing the directory content\n");

dir=opendir(argv[1]); while((rddir=readdir(dir))!=NULL)

{

printf("%s\t\n",rddir->d\_name);

}

closedir(dir);

}

**OUTPUT:** RP

roshi.c

first.c

pk6.c

f2

abc

FILE1

**RESULT:**-Thus the program was executed and verified successfully

**EX.NO : 4) PROGRAM USING SYSTEM CALLS open( ), read() & write()**

**AIM : T**o write the program to implement the system calls open( ),read( ) and write( ).

**ALGORITHM :**

Step 1 : Declare the structure elements.

Step 2 : Create a temporary file named temp1.

Step 3 : Open the file named “test” in a write mode.

Step 4 : Enter the strings for the file.

Step 5 : Write those strings in the file named “test”.

Step 6 : Create a temporary file named temp2.

Step 7 : Open the file named “test” in a read mode.

Step 8 : Read those strings present in the file “test” and save it in temp2.

Step 9 : Print the strings which are read.

**SYSTEM CALLS USED :**

**1.open( )**

**2.read( )**

**3.write( )**

**4.close( )**

**PROGRAM CODING**:

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

Void main( )

{

int fd1,fd2;

char buf[100];

fd1=open(“f1.txt”, O\_RDONLY); fd2=open(“f2.txt”, O\_WRONLY);

read(fd1,buf,size of(buf));

if(fd2!=-1)

{

write(fd2, buf, size of(buf));

printf(“Written”);

}

close(fd1);

close(fd2);

return 0;

}

**OUTPUT:**

**F1.txt :- Computer science F2.txt:-**

**Operating System**

**Written**

**F2.txt:- Computer science Operating System**

**RESULT:**-Thus the program was executed and verified successfully

**EX.NO: 5) CPU SCHEDULING: FIRST COME FIRST SERVE**

**AIM:** To write a c program to implement the first come first serve CPU scheduling algorithm

**PROBLEM DESCRIPTION:**

CPU scheduler will decide which process should be given the CPU for its execution.For this it uses different algorithm to choose among the process. One among that algorithm is FCFS algorithm. In this algorithm the process which arrive first is given the cpu after finishing its request only it will allow cpu to execute other process.

**ALGORITHM:**

Step 1: Create the number of process.

Step 2: Get the ID and Service time for each process.

Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 4: Calculate the Total time and Processing time for the remaining processes. Step 5: Waiting time of one process is the Total time of the previous process.

Step 6: Total time of process is calculated by adding Waiting time and Service

time.

Step 7: Total waiting time is calculated by adding the waiting time for lack process.

Step 8: Total turn around time is calculated by adding all total time of each process.

Step 9: Calculate Average waiting time by dividing the total waiting time by total

number of process.

Step 10: Calculate Average turn around time by dividing the total time by the number of process.

Step 11: Display the result.

**PROGRAM CODING:**

#include<stdio.h>

int main()

{

int n,b[10],t=0,i,w=0,r=0,a=0;

float avg,avg1;

printf("\n Enter number of processes:"); scanf("%d",&n);

printf("\n Enter the burst times : \n"); for(i=1;i<=n;i++)

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

printf("\n Gantt chart "); for(i=1;i<=n;i++) printf("P%d\t",i);

printf("\n\nProcess BurstTime WaitingTime TurnaroundTime\n"); for(i=1;i<=n;i++)

{

t=t+w;

r=r+b[i];

printf("P%d\t\t%d\t\t%d\t\t%d\t\t\n",i,b[i],w,r);

w=w+b[i];

a=a+r;

}

avg=(float)t/n;

avg1=(float)a/n;

printf("\n Average WaitingTime is %f",avg);

printf("\n Average TurnaroundTime is %f\n",avg1);

return(0);

}

**OUTPUT:**

Enter number of processes : 3

Enter the burst times :

24

5

3

|  |  |  |  |
| --- | --- | --- | --- |
| Gantt chart | P1 | P2 | P3 |
|  |  |  |  |
| Process | BurstTime | WaitingTime | TurnaroundTime |
| P1 | 24 | 0 | 24 |
| P2 | 5 | 24 | 29 |
| P3 | 3 | 29 | 32 |

Average WaitingTime is=17.666666

Average TurnaroundTime is=28.333334

**RESULT:**-Thus the program was executed and verified successfully

**EX.NO: 6) CPU SCHEDULING: SJF SCHEDULING**

**AIM:** To write a c program to implement the SJF CPU scheduling algorithm

**ALGORITHM:**

Step 1: Create the number of process.

Step 2: Get the ID and Service time for each process.

Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 4: Calculate the Total time and Processing time for the remaining processes. Step 5: Waiting time of one process is the Total time of the previous process.

Step 6: Total time of process is calculated by adding Waiting time and Service

time.

Step 7: Total waiting time is calculated by adding the waiting time for lack process.

Step 8: Total turn around time is calculated by adding all total time of each process.

Step 9: Calculate Average waiting time by dividing the total waiting time by total

number of process.

Step 10: Calculate Average turn around time by dividing the total time by the number of process.

Step 11: Display the result.

**PROGRAM CODING:**

#include<stdio.h>

int main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp,temp1;

float wtavg, tatavg;

printf("\n Enter the number of processes -- ");

scanf("%d", &n);

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

{

p[i]=i;

printf("Enter Burst Time for Process %d -- ", i);

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

}

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

{

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

{

if(bt[i]>bt[k])

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp1=p[i];

p[i]=p[k];

p[k]=temp1;

}

}

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

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

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

return(0);

|  |  |  |  |
| --- | --- | --- | --- |
| } |  |  |  |
| ***INPUT*** |  |  |  |
| Enter the number of processes -- | | 4 |  |
| Enter Burst Time for Process 0 -- | | 6 |  |
| Enter Burst Time for Process 1 -- | | 8 |  |
| Enter Burst Time for Process 2 -- | | 7 |  |
| Enter Burst Time for Process 3 -- | | 3 |  |
| ***OUTPUT*** |  |  |  |
| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
| P3 | 3 | 0 | 3 |
| P0 | 6 | 3 | 9 |
| P2 | 7 | 9 | 16 |
| P1 | 8 | 16 | 24 |
| Average Waiting Time -- | | 7.000000 |  |
| Average Turnaround Time -- | | 13.000000 |  |

**RESULT:**-Thus the program was executed and verified successfully

**EX.NO: 7)CPU SCHEDULING: PRIORITY SCHEDULING**

**AIM:** To write a C program to implement the CPU scheduling algorithm for Priority.

**PROBLEM DESCRIPTION:**

Cpu scheduler will decide which process should be given the CPU for its execution. For this it uses different algorithm to choose among the process. One among that algorithm is Priority algorithm. In this algorithm the processes will be given the priorities. The process which is having the highest priority is allocated the cpu first.

After finishing the request the cpu is allocated to the next highest priority and so on.

**ALGORITHM:**

Step 1: Get the number of process

Step 2: Get the id and service time for each process.

Step 3: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step 4: Calculate the total time and waiting time of remaining process.

Step 5: Waiting time of one process is the total time of the previous process.

Step 6: Total time of process is calculated by adding the waiting time and service time of each process.

Step 7: Total waiting time calculated by adding the waiting time of each process.

Step 8: Total turn around time calculated by adding all total time of each process.

Step 9: Calculate average waiting time by dividing the total waiting time by total

number of process.

Step 10: Calculate average turn around time by dividing the total waiting time by total number of process.

Step 11: Display the result.

**PROGRAM CODING:**

#include<stdio.h>

int main()

{

int n,bt[20]wt[20],p[20],tat[20],i,k,temp,temp1,temp2,pi[20];

float wtavg,tatavg;

printf("\nEnter no. of processes:"); scanf("%d",&n);

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

{

p[i]=i;

printf("\nEnter the prority:"); scanf("%d",&pi[i]);

printf("\nEnter the burst times : ");

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

}

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

{

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

{

if(pi[i]>pi[k])

{

temp=pi[i];

pi[i]=pi[k];

pi[k]=temp;

temp1=bt[i];

bt[i]=bt[k];

bt[k]=temp1;

temp2=p[i];

p[i]=p[k];

p[k]=temp2;

}

}

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\nProcess \t Priority\tBurst Time\t Waiting Time\t Turnaround Time"); for(i=1;i<=n;i++)

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

printf("\nAverage Waiting time =%f",wtavg);

printf("\nAverage Turnaround time = %f",tatavg);

return(0);

}

**OUTPUT:**

Enter the no. of processes : 3

Enter the burst times

P1 : 24

P2 : 5

P3 : 3

Enter the priorities

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 : 2 |  |  |  |  |
| P2 : 1 |  |  |  |  |
| P3 : 3 |  |  |  |  |
| ProcessID Priority | | BurstTime | WaitingTime TurnaroundTime | |
| P2 | 1 | 5 | 0 | 5 |
| P1 | 2 | 24 | 5 | 29 |
| P3 | 3 | 3 | 29 | 32 |

Average Waiting Time : 11.33

Average Turnaround Time : 22.00

**RESULT:** Thus the program was executed successfully.

**EXP NO.8) CPU SCHEDULING:ROUND ROBIN SCHEDULING**

**AIM:** To write a C program to implement the CPU scheduling algorithm for round robin.

**ALGORITHM:**

Step 1: Get the number of process

Step 2: Get the id and service time for each process.

Step 3: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step 4: Calculate the total time and waiting time of remaining process.

Step 5: Waiting time of one process is the total time of the previous process.

Step 6: Total time of process is calculated by adding the waiting time and service time of each process.

Step 7: Total waiting time calculated by adding the waiting time of each process.

Step 8: Total turn around time calculated by adding all total time of each process.

Step 9: Calculate average waiting time by dividing the total waiting time by total

number of process.

Step 10: Calculate average turn around time by dividing the total waiting time by total number of process.

Step 11: Display the result.

**PROGRAM CODING:**

#include<stdio.h>

int main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;

float awt=0,att=0,temp=0;

printf("Enter the no of processes -- ");

scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d -- ", i+1);

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

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- ");

scanf("%d",&t);

max=bu[0];

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

{

if(max<bu[i])

{

max=bu[i];

}

for(j=0;j<(max/t)+1;j++)

{

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

{

if(bu[i]!=0)

{

if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;

temp=temp+t;

}

}

}

}

}

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

{

wa[i]=tat[i]-ct[i];

att+=tat[i];

awt+=wa[i];

}

printf("\nAverage Waiting Time is --- %f",awt/n);

printf("\nAverage Turnaround Time is --- %f",att/n);

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i+1, ct[i], wa[i], tat[i]);

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| ***INPUT*** |  |  |  |  |
| Enter the number of processes -- 5 | | |  |  |
| Enter the Burst Time & Priority of Process 0 --- 10 | | | 3 |  |
| Enter the Burst Time & Priority of Process 1 --- 1 | | | 1 |  |
| Enter the Burst Time & Priority of Process 2 --- 2 | | | 4 |  |
| Enter the Burst Time & Priority of Process 3 --- 1 | | | 5 |  |
| Enter the Burst Time & Priority of Process 4 --- 5 | | | 2 |  |
| ***OUTPUT*** |  |  |  |  |
| PROCESS | PRIORITY | BURST TIME | WAITING TIME | TURNAROUND TIME |
| 1 | 1 | 1 | 0 | 1 |
| 4 | 2 | 5 | 1 | 6 |
| 0 | 3 | 10 | 6 | 16 |
| 2 | 4 | 2 | 16 | 18 |
| 3 | 5 | 1 | 18 | 19 |
| Average Waiting Time is --- | | 8.200000 |  |  |
| Average Turnaround Time is | | --- 12.000000 |  |  |

**RESULT:** Thus the program was executed successfully.

**EX.NO: 9 IMPLEMENTATION OF SEMAPHORE**

**AIM:** To write a C program to implement the Producer & consumer Problem(Semaphore)

**PROGRAM CODING**

#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\n2.consumer\n3.exit\n"); while(1)

{

printf(" \nenter ur choice"); scanf("%d",&n);

switch(n)

{

case 1:if((mutex==1)&&(empty!=0)) producer();

else

printf("buffer is full\n"); break;

case 2:if((mutex==1)&&(full!=0)) consumer();

else

printf("buffer is empty"); break;

case 3:exit(0);

break;

}

}

}

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 items %d",x);

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

printf("\n consumer consumes the item %d",x);

x--;

mutex=signal(mutex);

}

**RESULT:-**Thus the program was executed successfully

**OUTPUT:**

Produced element a Consumed element a Produced element b Consumed element b Produced element c Consumed element c Produced element d Consumed element d Produced element e Consumed element e Produced element f Consumed element f Produced element g Consumed element g Produced element h Consumed element h.

**EXP NO.10) Write a c program to implement paging with multiprogramming with a fixed number of task.**

**AIM:** To write a C program to implement PAGING with Multiprogramming with a Fixed number of Tasks

**PROGRAM CODING**

#include<stdio.h>

int main()

{

int ms, bs, nob, ef,n, mp[10],tif=0;

int i,p=0;

printf("Enter the total memory available (in Bytes) -- ");

scanf("%d",&ms);

printf("Enter the block size (in Bytes) -- ");

scanf("%d", &bs);

nob=ms/bs;

ef=ms - nob\*bs;

printf("\nEnter the number of processes -- ");

scanf("%d",&n);

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

{

printf("Enter memory required for process %d ",i+1); scanf("%d",&mp[i]);

}

printf("\nNo. of Blocks available in memory -- %d",nob);

printf("\n\nPROCESS\tMEMORY REQUIRED\t ALLOCATED\tINTERNAL FRAGMENTATION");

for(i=0;i<n && p<nob;i++)

{

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

if(mp[i] > bs)

printf("\t\tNO\t\t---");

else

{

printf("\t\tYES\t%d",bs-mp[i]);

tif = tif + bs-mp[i];

p++;

}

}

if(i<n)

printf("\nMemory is Full, Remaining Processes cannot be accomodated");

printf("\n\nTotal Internal Fragmentation is %d",tif);

printf("\nTotal External Fragmentation is %d",ef);

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***INPUT*** |  |  |  |  |  |
| Enter the total memory available (in Bytes) -- | | | | 1000 |  |
| Enter the block size (in Bytes)-- | | 300 |  |  |  |
| Enter the number of processes – 5 | |  |  |  |  |
| Enter memory required for process 1 (in Bytes) -- | | | | 275 |  |
| Enter memory required for process 2 (in Bytes) -- | | | | 400 |  |
| Enter memory required for process 3 (in Bytes) -- | | | | 290 |  |
| Enter memory required for process 4 (in Bytes) -- | | | | 293 |  |
| Enter memory required for process 5 (in Bytes) -- | | | | 100 |  |
| No. of Blocks available in memory -- | | | 3 |  |  |
| ***OUTPUT*** |  |  |  |  |  |
| PROCESS | MEMORY REQUIRED | | | ALLOCATED | INTERNAL FRAGMENTATION |
| 1 | 275 |  |  | YES | 25 |
| 2 | 400 |  |  | NO | ----- |
| 3 | 290 |  |  | YES | 10 |
| 4 | 293 |  |  | YES | 7 |
| Memory is Full, Remaining Processes cannot be accommodated | | | | |  |
| Total Internal Fragmentation is | | 42 |  |  |  |
| Total External Fragmentation is | | 100 |  |  |  |

**RESULT:-**Thus the program was executed successfully

**EX.NO: 11)** **To write a C program to implement PAGING with Multiprogramming with a variable number of task**

**AIM:** To write a C program to implement PAGING with Multiprogramming with a variable number of task

**PROGRAM CODING**

#include<stdio.h>

int main()

{

int ms,mp[10],i, temp,n=0;

char ch = 'y';

printf("\nEnter the total memory available (in Bytes)-- ");

scanf("%d",&ms);

temp=ms;

for(i=0;ch=='y';i++,n++)

{

printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);

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

if(mp[i]<=temp)

{

printf("\nMemory is allocated for Process %d ",i+1);

temp = temp - mp[i];

}

Else

{

printf("\nMemory is Full");

break;

}

printf("\nDo you want to continue(y/n) -- ");

scanf(" %c", &ch);

}

printf("\n\nTotal Memory Available -- %d", ms);

printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED ");

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

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

printf("\n\nTotal Memory Allocated is %d",ms-temp);

printf("\nTotal External Fragmentation is %d",temp);

return(0);

}

|  |  |  |  |
| --- | --- | --- | --- |
| ***INPUT*** |  |  |  |
| Enter the total memory available (in Bytes) -- | | | 1000 |
| Enter memory required for process 1 (in Bytes) -- | | | 400 |
| Memory is allocated for Process 1 | |  |  |
| Do you want to continue(y/n) -- | | y |  |
| Enter memory required for process 2 (in Bytes) -- | | | 275 |
| Memory is allocated for Process 2 | |  |  |
| Do you want to continue(y/n) -- | | y |  |
| Enter memory required for process 3 (in Bytes) -- | | | 550 |
| ***OUTPUT*** |  |  |  |
| Memory is Full |  |  |  |
| Total Memory Available -- 1000 | |  |  |
| PROCESS | MEMORY ALLOCATED | |  |
| 1 | 400 |  |  |
| 2 | 275 |  |  |
| Total Memory Allocated is 675 | |  |  |
| Total External Fragmentation is | | 325 |  |

**RESULT:-**Thus the program was executed successfully

**EX.NO: 12****)Write a C program to simulate page replacement algorithm FIFO**

**AIM:** To write a C program to implement FIFO page replacement algorithm.

**PROGRAM CODING**

#include<stdio.h>

int main()

{

int i, j, k, f, pf=0, count=0, rs[25], m[10], n;

printf("\n Enter the length of reference string -- ");

scanf("%d",&n);

printf("\n Enter the reference string -- ");

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

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

printf("\n Enter no. of frames -- ");

scanf("%d",&f);

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

m[i]=-1;

printf("\n The Page Replacement Process is -- \n");

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

{

for(k=0;k<f;k++)

{

if(m[k]==rs[i])

break;

}

if(k==f)

{

m[count++]=rs[i];

pf++;

}

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

printf("\t%d",m[j]);

if(k==f)

printf("\tPF No. %d",pf);

printf("\n");

if(count==f)

count=0;

}

printf("\n The number of Page Faults using FIFO are %d",pf);

return(0);

}

***INPUT***

Enter the length of reference string – 20

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Enter the reference string -- | | | | 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 |
| Enter no. of frames -- | | | 3 |  |
| ***OUTPUT*** |  |  |  |  |
| The Page Replacement Process is – | | | |  |
| 7 | -1 | -1 | PF No. 1 |  |
| 7 | 0 | -1 | PF No. 2 |  |
| 7 | 0 | 1 | PF No. 3 |  |
| 2 | 0 | 1 | PF No. 4 |  |
| 2 | 0 | 1 |  |  |
| 2 | 3 | 1 | PF No. 5 |  |
| 2 | 3 | 0 | PF No. 6 |  |
| 4 | 3 | 0 | PF No. 7 |  |
| 4 | 2 | 0 | PF No. 8 |  |
| 4 | 2 | 3 | PF No. 9 |  |
| 0 | 2 | 3 | PF No. 10 | |
| 0 | 2 | 3 |  |  |
| 0 | 2 | 3 |  |  |
| 0 | 1 | 3 | PF No. 11 | |
| 0 | 1 | 2 | PF No. 12 | |
| 0 | 1 | 2 |  |  |
| 0 | 1 | 2 |  |  |
| 7 | 1 | 2 | PF No. 13 | |
| 7 | 0 | 2 | PF No. 14 | |
| 7 | 0 | 1 | PF No. 15 | |

The number of Page Faults using FIFO are 15

**RESULT:-**Thus the program was executed successfully.