EX.NO: 01

DATE: 24/08/22

SHORTEST JOB FIRST ALGORITHM

AIM:

To write a c program to implement the shortest job first algorithm.

PROCEDURE:

Step 1 : Start the process.

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

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

Step 4 : Start the Ready Queue 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.

(a) Waiting Time for process = Turnaround Time - Burst time

(b) Turnaround time for process = waiting time of process + Burst time for process.

Step 7: calculate the following

(a) Average waiting time = Total waiting time / Number of process

(b) Average Turnaround time = Total turnaround time / Number of process

Step 8 : Stop the process.

RESULT:

Thus, the C program to implement the shortest job first algorithm has been executed successfully.

SHORTEST JOB FIRST ALGORITHM

```
#include<stdio.h>
 void main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
   float avg_wt,avg_tat;
   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];
      total+=wt[i];
    }
    avg_wt=(float)total/n;
    total=0;
    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
      tat[i]=bt[i]+wt[i];
      total+=tat[i];
      printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=(float)total/n;
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

OUTPUT:

Enter number of process:3

Enter Burst Time:

p1:12

p2:12

p3:11

Process	Burst Time	Waiting Time	Turnaround Time
p3	11	0	11
p2	12	11	23
p1	12	23	35

Average Waiting Time=11.333333

Average Turnaround Time=23.000000

EX.NO: 02

DATE: 08/09/22

FIRST COME FIRST SERVED ALGORITHM

AIM:

To write a c program to implement the first come first served algorithm.

PROCEDURE:

Step 1 : Start the process

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

Step 3 : For each process in the ready Queue, 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 turnaround time.

Step 5 : For each process in the Ready queue, calculate

(a) Waiting Time for process = Turnaround Time - Burst Time of process

(b) Turnaround time for process = waiting Time + Burst Time of process.

Setep 6 : Calculate the following

(a) Average waiting Time - Total waiting Time /
Number of process

(b) Average Turnaround Time - total Turnaround Time /
Number of process

Step 8 : Stop the process.

RESULT :

Thus, the C program to implement the first come first server Algorithm has been executed successfully.

FIRST COME FIRST SERVED ALGORITHM

```
#include<stdio.h>
int main()
{
int n,bt [20], wt [20], tat [20], avwt=0, avtat=0,i,j;
printf("Enter total number of processes (maximum 20):");
scanf("%d", &n);
printf("\nEnter Process Burst Time\n");
for (i=0;i<n;i++)
{
printf("P[%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("\nP[%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;
}
```

OUTPUT:

Enter total number of processes (maximum 20):3

Enter Process Burst Time

P[1]:11

P[2]:20

P[3]:21

Process	Burst Time	Waiting Time	Turnaround Time
P[1]	11	0	11
P[2]	20	11	31
P[3]	21	31	52

Average Waiting Time: 14

Average Turnaround Time: 31

EX.NO: 03

DATE: 26/09/22

IMPLEMENTATION OF READER-WRITER PROBLEM USING SEMAPHORE

AIM:

To write a c program to implement the Reader-writer Problem using semaphore.

PROCEDURE:

Reader Process:

Step 1: Start the process

Step 2: Reader requests the entry to critical section.

Step 3: If allowed:

It increments the count of number of reader inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any readers is inside.

If then signals mutex as any other reader is allowed to enter while others are already reading.

After performing reading, it exists the critical section. when exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as

now, writer can enter the critical section.

Step 4: If not allowed, it keeps on waiting

Step 5: Stop the process.

writer process

Step 1: start the process

Step 2: writer requests the entry to critical section

Step 3: If allowed (1.e) wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.

1. It exists the critical section.

RESULT:-

Thus, the C program to implement the Readers / writers problem using semaphore has been executed successfully.

TO IMPLEMENT READER/WRITER PROBLEM USING SEMAPHORE

```
#include<stdio.h>
#include<conio.h>
int x=1,rc=0,readcount=1;
void p (int*a)
{
   while(*a==0)
   {
      printf("Busy Wait");
   }
*a=*a-1;
}
void v(int*b)
{
   *b=*b+1;
}
void p1(int*c)
{
   while(*c==0)
   {
      printf("Busy Wait");
   }
*c=*c-1;
}
void v1(int*d)
{
   *d=*d+1;
}
void reader()
{
   int flag=1;
```

```
while(flag==1)
{
   p(&readcount);
   rc=rc+1;
   if(rc==1)
   p1(&x);
   v(&readcount);
   printf("\nReader is reading");
   p(&readcount);
   rc=rc-1;
   if(rc==0)
   v1(&x);
   v(&readcount);
   flag=0;
  }
}
void writer()
{
  p1(&x);
  printf("\nWriter is writing");
  v1(&x);
}
void main()
{
   printf("\nReaders-Writers Problem");
   printf("\n**********************\n");
   reader();
   writer();
   reader();
   writer();
   getch();
}
```

OUTPUT:

Readers-Writers Problem

****

Reader is reading

Writer is writing

Reader is reading

Writer is writing

EX.NO: 04

DATE: 11/10/22

IMPLEMENTATION OF FIFO PAGE REPLACEMENT ALGORITHM

AIM:

To write a C program to implement the FIFO Page Replacement algorithm.

PROCEDURE:

Step 1: Start the process.

Step 2: Declare the size with respect to page length.

Step 3: check the need of replacement from the page to memory.

Step 4: check the need of replacement from old page to new page in memory.

Step 5: Form a queue to hold all pages.

Step 6: Insert the page require memory into the queue.

Step 7: check for bad replacement and page fault.

Step 8: Get the number of processes to be inserted.

Step 9: Display the values.

Step 10: Stop the process.

RESULT:

Thus, the C program to implement the FIFO page Replacement algorithm has been executed successfully.

FIRST IN FIRST OUT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,j,n,a[50],frame[10],no,k,avail,count=0;

 printf("\nFIFO PAGE REPLACEMENT ALGORITHM");
 printf("\n*************************************");
 printf("\nENTER THE NUMBER OF PAGES: ");
 scanf("%d",&n);
 printf("ENTER THE PAGE NUMBER: ");
 for(i=1;i<=n;i++)
 {
 scanf("%d",&a[i]);
 }
 printf("ENTER THE NUMBER OF FRAMES: ");
 scanf("%d",&no);
 for(i=0;i<no;i++)
 frame[i]= -1;
 j=0;
 printf("Ref String Page Frames\n");
 for(i=1;i<=n;i++)
 {
 printf("%d\t\t",a[i]);
 avail=0;
 for(k=0;k<no;k++)
 if(frame[k]==a[i])
 avail=1;
 if(avail==0)
```

```
 {
frame[j]=a[i];
j=(j+1)%no;
count++;
for(k=0;k<no;k++)
 printf("%d\t",frame[k]);
 }
 printf("\n");
 }
 printf("Page Fault: %d",count);
 getch();
}
```

OUTPUT:

FIFO PAGE REPLACEMENT ALGORITHM

****

ENTER THE NUMBER OF PAGES: 20

ENTER THE PAGE NUMBER: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

ENTER THE NUMBER OF FRAMES: 2 3

Ref String Page Frames

7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1
0			
3	2	3	1
0	2	3	0
4	4	3	0
2	4	2	0
3	4	2	3
0	0	2	3
3			
2			
1	0	1	3
2	0	1	2
0			
1			
7	7	1	2
0	7	0	2
1	7	0	1

Page Fault: 15

EX.NO: 05

DATE: 26/10/22

PROGRAM FOR INTER-PROCESS COMMUNICATION

AIM:

To write a C Program to implement inter-process communication using shared memory.

PROCEDURE:

Step 1: Start the process

Step 2: Declare the segment size

Step 3: Create the shared memory

Step 4: Read the data from the shared memory

Step 5: Write the data to the shared memory

Step 6: Edit the data

Step 7: Stop the process

RESULT:

Thus, the C program to implement inter-process communication using shared memory has been executed successfully.

PROGRAM FOR INTER-PROCESS COMMUNICATION

```
#include<stdio.h>
#include<conio.h>
struct segtab
{
 int limit,base;
}st[50];
int main()
{
 int n,i,segno,offset,paddr;
 printf("\nInter-Process Communication Using Shared Memory");
 printf("\n*****************************************");
 printf("\nEnter the no of segments: ");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("\nEnter the base for segment %d: ",i);
 scanf("%d",&st[i].base);
 printf("\nEnter the limit for the segment %d: ",i);
 scanf("%d",&st[i].limit);
 }
 printf("\nEnter the segment no and offset to produce physical address: ");
 scanf("%d%d",&segno,&offset);
 if(segno>=n)
 {
 printf("\nSegment number invalid");
 getch();
 return 0;
 }
 if(offset>st[segno].limit)
```

```
 {
 printf("\nOffset is not in the limit");
 getch();
 return 0;
 }
 paddr=st[segno].base+offset;
 printf("\nPhysical address for the given segment is: %d",paddr);
 printf("\nEnter the segment no and offset to produce physical address: ");
 scanf("%d%d",&segno,&offset);
 if(segno>=n)
 {
printf("\nSegment number invalid");
getch();
return 0;
 }
 if(offset>st[segno].limit)
 {
 printf("\nOffset is not in the limit");
 getch();
 return 0;
 }
 getch();
 return 0;
}
```

OUTPUT:

Inter-Process Communication Using Shared Memory

****

Enter the no of segments: 5

Enter the base for segment 0: 1400

Enter the limit for the segment 0: 1000

Enter the base for segment 1: 6300

Enter the limit for the segment 1: 400

Enter the base for segment 2: 4300

Enter the limit for the segment 2: 400

Enter the base for segment 3: 4 3200

Enter the limit for the segment 3: 400

Enter the base for segment 4: 4700

Enter the limit for the segment 4: 1000

Enter the segment no and offset to produce physical address: 4750 4 750

Physical address for the given segment is: 5450

Enter the segment no and offset to produce physical address: 1 500

Offset is not in the limit