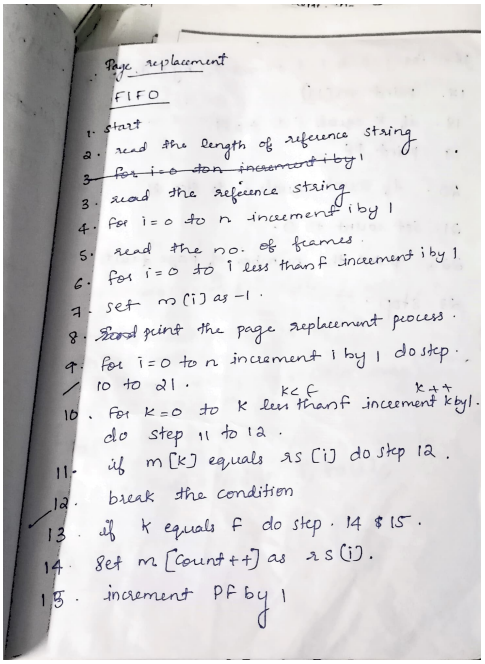
```
#include <stdio.h>#include<std lib.h>
int mutex=1;
int ful l=0;
int empty=5,x=0;
void producer()
{
--mutex;
++ful l;
--empty;
x++;
printf("\n producer producers" "item %d",x);
++mutex;
}
void consumer()
{
--mutex;
--ful l;
++empty;
x--;
printf("\n consumer consumes" "item %d",x);
++mutex;
}
int main()
{
int n,i;
printf("/n press 1 for producer,press 2 for
consumer,press 3 for exit");
for(i=1;i>0;i++)
{
printf("\n enter your choice");
scanf("%d",&n);
switch(n)
```

```
{
case 1:if((mutex==1)&&(empty!=0))
{
producer();
}
else
{
printf("buffer is ful l");
}
break;
case 2:if((mutex==1)&&(ful l!=0))
{
consumer();
}
else
{
printf("buffer is empty");
}
break;
case 3:
exit(0);
}
}
}
```

Algorithm for FCFS and SCAN Disk Scheduling using Switch Case

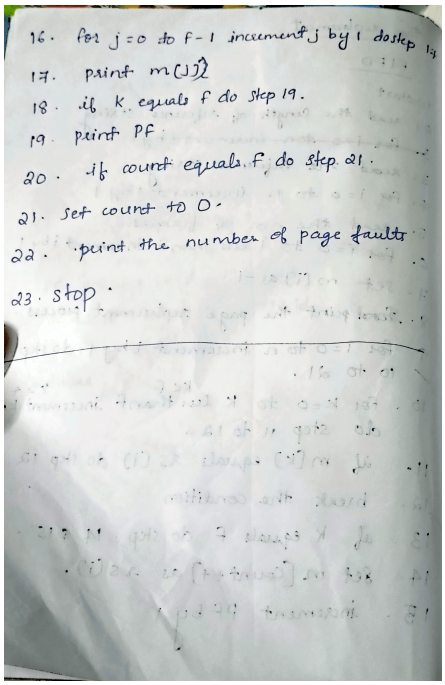
1. Start
2. Declare variables: n, head, disk_size, choice, and an integer array requests[]
3. Input the number of disk requests
4. Create an array of size n to store the disk request sequence
5. Input the request sequence from the user
    5.1 Store each request into the array
6. Input the initial position of the disk head
7. Input the total size of the disk
8. Display algorithm options to the user
    8.1 Option 1 → FCFS
    8.2 Option 2 → SCAN
9. Take user's choice as input
10. Use switch-case to perform the selected scheduling
    10.1 If choice = 1, call FCFS function
    10.2 If choice = 2, call SCAN function
    10.3 If choice is invalid, print error message
11. In FCFS function:
    11.1 Declare and initialize seek_time = 0
    11.2 Print initial head position
    11.3 For each request in order:
        11.3.1 Calculate absolute difference between head and current request
        11.3.2 Add difference to seek_time
        11.3.3 Update head to current request
    11.4 Print total seek time and movement sequence
12. In SCAN function:
    12.1 Create a new array sorted_requests[] of size n + 2
    12.2 Add 0 and disk_size - 1 as boundary values to the array
    12.3 Copy all disk requests into the array
    12.4 Sort the array in ascending order
    12.5 Find the position where head fits in the sorted array
    12.6 Move head from current position to the highest track
        12.6.1 For each movement, update seek_time and head
    12.7 After reaching the end, move head back toward 0
        12.7.1 For each movement, update seek_time and head
    12.8 Print total seek time and complete movement path
13. Stop



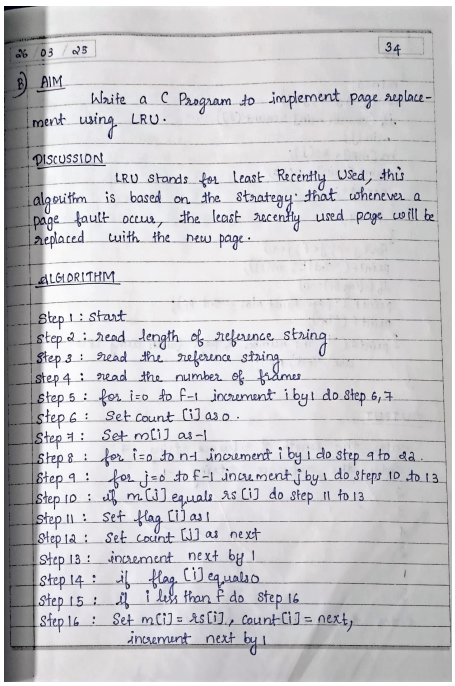
Page replacement

FIFO

1. start
2. read the length of reference string
3. ~~for i=0 ton increment i by 1~~
3. read the reference string
4. for i = 0 to n increment i by 1
5. read the no. of frames
6. for i = 0 to 1 less than f increment i by 1
7. set m[i] as -1
8. ~~Start~~ print the page replacement process.
9. for i = 0 to n increment i by 1 do step 10 to 21.
10. For k = 0 to k less than f ($k<f$) increment k by 1 ($k++$) do step 11 to 12.
11. if m[k] equals rs[i] do step 12.
12. break the condition
13. if k equals f do step 14 & 15.
14. Set m[count++] as rs[i].
15. increment PF by 1
16. for j = 0 to f-1 increment j by 1 do step 17
17. print m[j]
18. if k equals f do step 19.
19. print PF
20. if count equals f do step 21.
21. Set count to 0.
22. print the number of page faults
23. stop.



```
#include <stdio.h>
void main()
{
    int i, j, k, f, n, rs[20], m[10], count=0, pf=0;
    printf("enter the lenght of reference string--");
    scanf("%d", &n);
    printf("enter the reference string--");
    for(i=0; i<n; i++)
    scanf("%d", &rs[i]);
    printf("enter the no of frames--");
    scanf("%d", &f);
    for(i=0; i<f; i++)
    m[i]=-1;
    printf("\n the page replacement process--\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("\t pf no %d", pf);
        printf("\n");
        if(count==f)
        count=0;
    }
    printf("\n the number of page fault using FIFO are %d", pf);
}
```



06 03 25 34

B) AIM

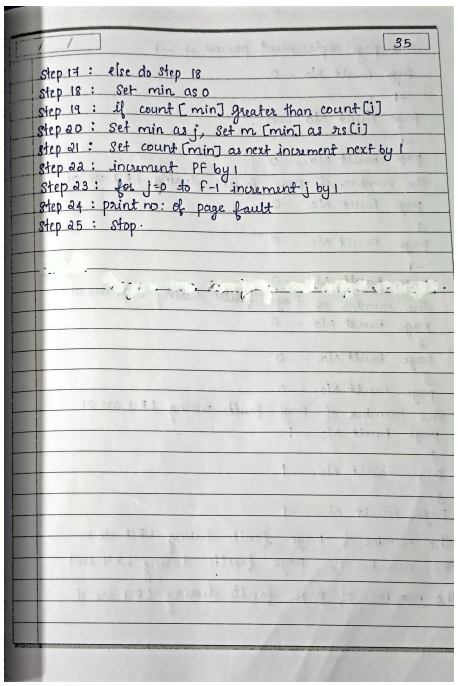
Write a C Program to implement page replacement using LRU.

DISCUSSION

LRU stands for Least Recently Used, this algorithm is based on the strategy that whenever a page fault occur, the least recently used page will be replaced with the new page.

ALGORITHM

Step 1 : Start
Step 2 : read length of reference string
Step 3 : read the reference string
Step 4 : read the number of frames
Step 5 : for i=0 to f-1 increment i by1 do Step 6,7
Step 6 : Set count [i] as 0.
Step 7 : Set m[i] as -1
Step 8 : for i=0 to n-1 increment i by 1 do step 9 to 22.
Step 9 : for j=0 to f-1 increment j by 1 do steps 10 to 13
Step 10 : if m[j] equals rs[i] do step 11 to 13
Step 11 : Set flag [i] as 1
Step 12 : Set count [j] as next
Step 13 : increment next by 1
Step 14 : if flag [i] equals 0
Step 15 : if i less than f do step 16
Step 16 : Set m[i] = rs[i], count[i] = next, increment next by 1

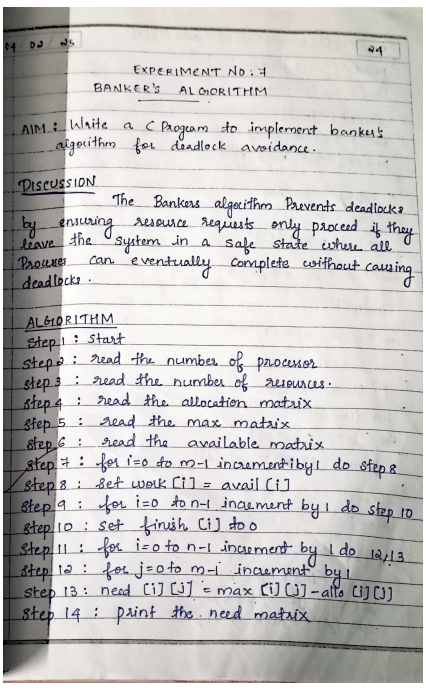


55

Step 17 : else do step 18
Step 18 : Set min as 0
Step 19 : if count [min] greater than count [j]
Step 20 : Set min as j, Set m [min] as rs[i]
Step 21 : Set count [min] as next increment next by 1
Step 22 : increment PF by 1
Step 23 : for j=0 to f-1 increment j by 1
Step 24 : print no: of page fault
Step 25 : Stop.

```
#include <stdio.h>
void main()
{
    int i, j, k, min, rs[20], m[10], count[10], flag[25], n, f, pf=0, next=1;
    printf("enter the lenght of reference string--");
    scanf("%d", &n);
    printf("enter the reference string--");
    for(i=0; i<n; i++)
    {
        scanf("%d", &rs[i]);
        flag[i]=0;
    }
    printf("enter the no of frames--");
    scanf("%d", &f);
    for(i=0; i<f; i++)
    {
        count[i]=0;
        m[i]=-1;
    }
    printf("\n the page replacement process--\n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<f; j++)
        {
            if(m[j]==rs[i])
            {
                flag[i]=1;
                count[j]=next;
                next++;
            }
        }
        if(flag[i]==0)
        {
            if(i<f)
            {
                m[i]=rs[i];
                count[i]=next;
                next++;
            }
            else
```

```
        {
            min=0;
            for(j=1; j<f; j++)
            if(count[min]>count[j])
            min=j;
            m[min]=rs[i];
            count[min]=next;
            next++;
        }
        pf++;
    }
    for(j=0; j<f; j++)
    printf("%d\t", m[j]);
    if(flag[i]==0)
    printf("page fault no, --%d", pf);
    printf("\n");
}
printf("\n the number of page faults using LRU are %d", pf);
```



24 02 25 24

EXPERIMENT NO : 7
BANKER'S ALGORITHM

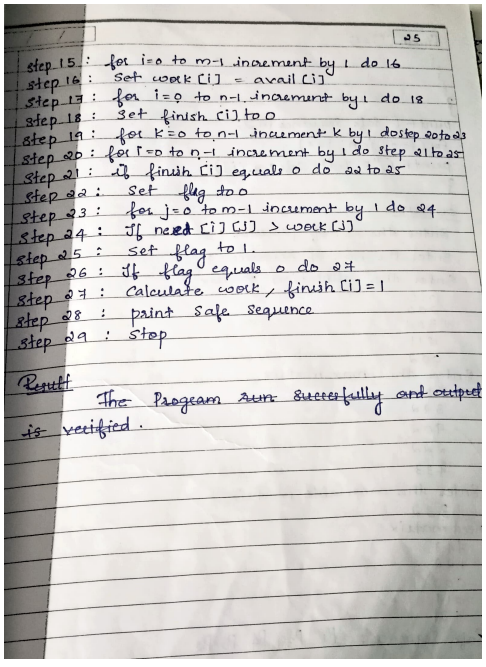
AIM : Write a C Program to implement banker's algorithm for deadlock avoidance.

DISCUSSION

The Bankers algorithm Prevents deadlocks by ensuring resource requests only proceed if they leave the system in a safe state where all Process can eventually complete without causing deadlocks.

ALGORITHM

Step 1 : Start
Step 2 : read the number of processor
Step 3 : read the number of resources.
Step 4 : read the allocation matrix
Step 5 : read the max matrix
Step 6 : read the available matrix
Step 7 : for i=0 to m-1 increment i by 1 do step 8
Step 8 : Set work [i] = avail [i]
Step 9 : for i=0 to n-1 increment by 1 do step 10
Step 10 : set finish [i] to 0
Step 11 : for i=0 to n-1 increment by 1 do 12,13
Step 12 : for j=0 to m-1 increment by 1
Step 13 : read [i] [j] = max [i] [j] - allo [i] [j]
Step 14 : print the need matrix



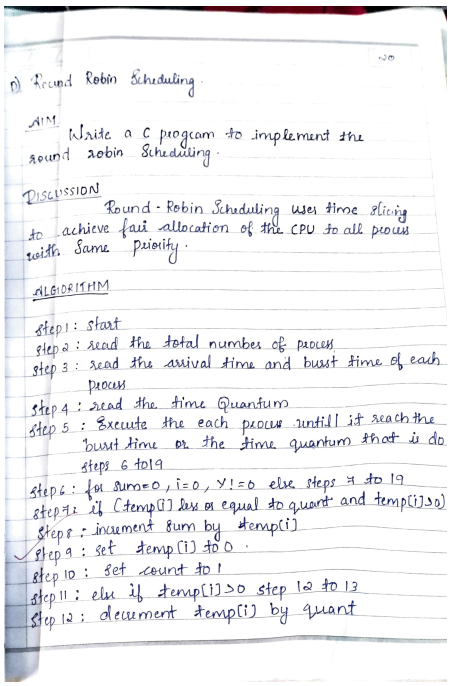
25

Step 15 : for i=0 to m-1 increment by 1 do 16
Step 16 : Set work [i] = avail [i]
Step 17 : for i=0 to n-1 increment by 1 do 18
Step 18 : Set finish [i] to 0
Step 19 : for k=0 to n-1 increment k by 1 do step 20 to 23
Step 20 : for i=0 to n-1 increment by 1 do step 21 to 23
Step 21 : if finish [i] equals 0 do 22 to 25
Step 22 : Set flag to 0
Step 23 : for j=0 to m-1 increment by 1 do 24
Step 24 : if need [i] [j] > work [j]
Step 25 : set flag to 1.
Step 26 : if flag equals 0 do 27
Step 27 : Calculate work, finish [i] = 1
Step 28 : print Safe Sequence
Step 29 : Stop

Result

The Progeam ~~run Succesfully and output is verified~~.

```
#include <stdio. h>
void main()
{
    int n, i, m, j, k, y, work[10], finish[10], ind=0;
    int allo[20][20], max[20][20], need[20][20], safeseq[20], avail[20];
    printf("enter the no of process");
    scanf("%d", &n);
    printf("enter the no of resources");
    scanf("%d", &m);
    printf("enter the allo matrix \n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<m; j++)
        scanf("%d", &allo[i][j]);
    }
    printf("enter the max matrix\n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<m; j++)
        scanf("%d", &max[i][j]);
    }
    printf("enter the avail matrix\n");
    for(i=0; i<m; i++)
    scanf("%d", &avail[i]);
    for(i=0; i<m; i++)
    work[i]=avail[i];
    for(i=0; i<n; i++)
    finish[i]=0;
    for(i=0; i<n; i++)
    {
        for(j=0; j<m; j++)
        need[i][j]=max[i][j]-allo[i][j];

    }
    printf("\n need matrix \n");
    for(i=0; i<n; i++)
    {
        printf("\n");
        for(j=0; j<m; j++)
        {
            printf("%d", need[i][j]);
        }
    }
    //safety algorithm
    for(i=0; i<m; i++)
    work[i]=avail[i];
    for(i=0; i<n; i++)
    finish[i]=0;
    for(k=0; k<n; k++)
    {
        for(i=0; i<n; i++)
        {
            if(finish[i]==0)
            {
                int flag=0;
                for(j=0; j<m; j++)
                {
                    if(need[i][j]>work[j])
                    {
                        flag=1;
                        break;
                    }
                }
                if(flag==0)
                {
                    safeseq[ind++]=i;
                    for(y=0; y<m; y++)
                    {
                        work[y]+=allo[i][y];
                        finish[i]=1;
                    }
                }
            }
        }

    }
    printf("\n the safe sequence:\n");
    for(i=0; i<n; i++)
    {
        printf("p%d", safeseq[i]);
    }
}
```



1) Round Robin Scheduling.

AIM

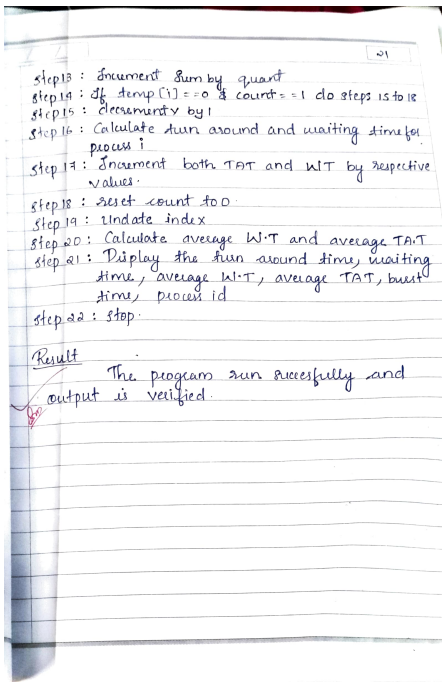
Write a C program to implement the round robin Scheduling.

DISCUSSION

Round-Robin Scheduling uses time slicing to achieve fair allocation of the CPU to all process with same priority.

ALGORITHM

Step 1 : Start
Step 2 : read the total numbes of process
Step 3 : read the arrival time and burst time of each process
Step 4 : read the time Quantum
Step 5 : Execute the each process until it reach the burst time or the time quantum that is do steps 6 to 19
Step 6 : for sum=0, i=0, Y!=0 else steps 7 to 19
Step 7 : if (temp[i] less or equal to quant and temp[i]>0)
Step 8 : increment sum by temp[i]
Step 9 : set temp[i] to 0.
Step 10 : set count to 1
Step 11 : else if temp[i]>0 step 12 to 13
Step 12 : decrement temp[i] by quant

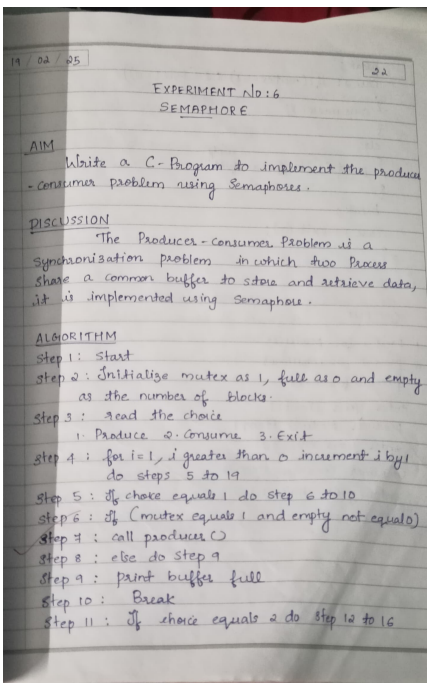


Step 13 : Increment sum by quant
Step 14 : If temp[i]==0 & count==1 do steps 15 to 18
Step 15 : decrement y by 1
Step 16 : Calculate turn around and waiting time for process i
Step 17 : Increment both TAT and WT by respective values.
Step 18 : reset count to 0
Step 19 : Update index
Step 20 : Calculate average W.T and average TAT
Step 21 : Diplay the turn around time, waiting time, average W.T, average TAT, burst time, process id
Step 22 : stop.

Result

The program run succesfully and output is verified.

```
#include<stdio. h>
void main()
{
int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
float avg_wt, avg_tat;
printf("Total number of process in the system");
scanf("%d", &NOP);
y=NOP;
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the process[%d]\n", i+1);
printf("arrival time is:");
scanf("%d", &at[i]);
printf("\n burst time is:");
scanf("%d", &bt[i]);
temp[i]=bt[i];
}
printf("Enter the quantum for the process:\t");
scanf("%d", &quant);
printf("\nprocess No\t burst time\tTAT \twaiting time");

while(y>0){
for(i=0; i<NOP; i++){

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("\nProcess No[%d]\t\t%d\t\t\t\t%d\t\t\t\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;
}
}
}
avg_wt=wt * 1.0 /NOP;
avg_tat=tat * 1.0 /NOP;
printf("\n average turn around time :%f" , avg_tat);
printf("\n average waiting time :%f" , avg_wt);
}
```



19/02/25

EXPERIMENT No:6
SEMAPHORE

AIM

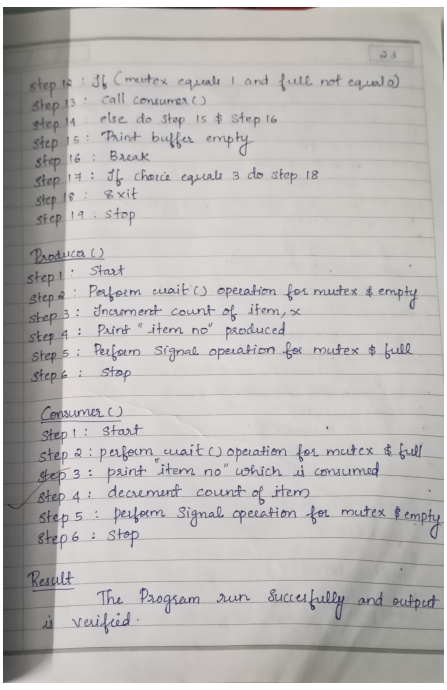
Write a C-Program to implement the producer-consumer problem using Semaphores.

DISCUSSION

The Producer-consumer Problem is a synchronisation problem in which two Process share a common buffer to store and retrieve data, it is implemented using Semaphore.

ALGORITHM

Step 1 : Start
Step 2 : Initialize mutex as 1, full as 0 and empty as the number of blocks.
Step 3 : read the choice
1. Produce 2. Consume 3. Exit
Step 4 : for i=1, i greater than 0 increment i by 1 do steps 5 to 19
Step 5 : If choice equals 1 do step 6 to 10
Step 6 : If (mutex equals 1 and empty not equal 0)
Step 7 : call producer()
Step 8 : else do step 9
Step 9 : print buffer full
Step 10 : Break
Step 11 : If choice equals 2 do step 12 to 16



Step 12 : If (mutex equals 1 and full not equal 0)
Step 13 : call consumer()
Step 14 : else do Step 15 & Step 16
Step 15 : Print buffer empty
Step 16 : Break
Step 17 : If choice equals 3 do step 18
Step 18 : Exit
Step 19 : stop

Producer()
Step 1 : Start
Step 2 : Perform wait() operation for mutex & empty
Step 3 : Increment count of item, x
Step 4 : Print "item no" produced
Step 5 : Perform Signal operation for mutex & full
Step 6 : Stop

Consumer()
Step 1 : Start
Step 2 : perform wait() operation for mutex & full
Step 3 : print "item no" which is consumed
Step 4 : decrement count of item
Step 5 : perform Signal operation for mutex & empty
Step 6 : Stop

Result

The Program run Succesfully and output is verified.

```
#include <stdio.h>
#include<stdlib.h>
int mutex=1;
int full=0;
int empty=5, x=0;
void producer()
{
    --mutex;
    ++full;
    --empty;
    x++;
    printf("\n producer producers" "item %d", x);
    ++mutex;
}
void consumer()
{
     --mutex;
     --full;
     ++empty;
     x--;
     printf("\n consumer consumes" "item %d", x);
     ++mutex;

}
int main()
{
    int n, i;
    printf("/n press 1 for producer, press 2 for consumer, press 3 for exit");
    for(i=1; i>0; i++)
    {
        printf("\n enter your choice");
        scanf("%d", &n);
        switch(n)
        {
            case 1:if((mutex==1)&&(empty!=0))
            {
                producer();
            }
            else
            {
                printf("buffer is full");
            }
            break;
            case 2:if((mutex==1)&&(full!=0))
            {
                consumer();
            }
            else
            {
                printf("buffer is empty");
            }
            break;
            case 3:
            exit(0);
        }
    }
}
```

Buoxcity

1. start
2. Declare the variables
3. read the number of process, n
4. For i=0 to n increment i by 1 do step 5
   5. read the burst time for each process, bt[i]
6. For i=0 to n increment i by 1 do step 7
   Step7: read the priority of each process, P[i]
8. For i=0 to n-1 increment i by 1 do step 9 to 16
   9. For j=0 to n-i-1 increment j by 1 do step 10 to 16
      10. If ((P[j] > P[j+1]) && (bt[j] < bt[j+1])) do step 11 to 16
          11. Set temp1 = bt[j];
          12. Set bt[j] = bt[j+1];
          13. Set bt[j+1] = temp;
          14. Set temp2 = p[j];
          15. Set P[j] = P[j+1]
          16. Set P[j+1] = temp2;
17. Set W[i]=0
18. For i=0 to n increment i by 1 do step 19
    19. W[i] = W[i-1] + bt[i-1]
20. For i=0 to n increment i by 1 do step 21
    21. Display the waiting time for each process, W[i]
22. For i=0 to n increment i by 1 do step 23 & 24
    23. Set tsum = tsum + w[i]
    24. Set ta[i] = ta[i] + W[i] + bt[i]
25. For i=0 to n increment i by 1 do step 26
    26. Display the turnaround time for each process ta[i]
27. For i=0 to n increment i by 1 do step 28
    28. tSum = tSum + ta[i]
29. Calculate Average TAT & Avg WT and display it
30. Stop

wsum → total WT
tsum → total TAT
wavg → W.T Avg
tavg → TAT Avg

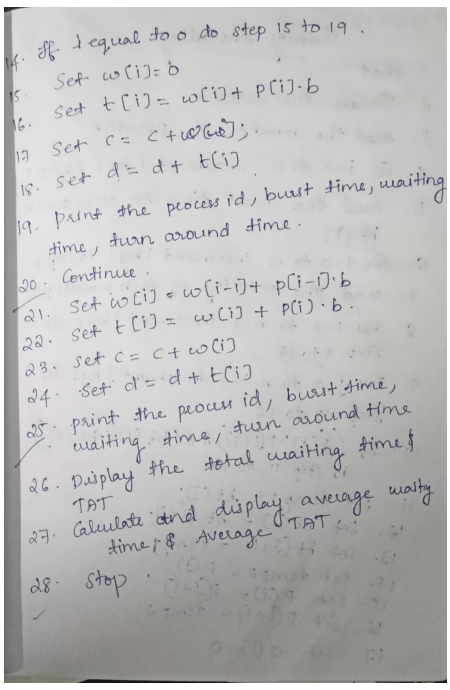
ta[20], fo j

TAT = sum2
total WT = sum1

```
#include <stdio.h>
void main()
{
   int n, i, j, pt[20], bt[20], wt[20], ta[20]={0};
   float wavg=0,  tavg=20;
   int wsum, tsum, temp1, temp2;
   printf("enter the number of process:\n");
   scanf("%d", &n);
   printf("enter the burst time for each processes:\n");
   for(i=0; i<n; i++)
   {
      printf("burst time for %dth process\n", i);
      scanf("%d", &bt[i]);
   }
   printf("enter the priorites:\n");
   for(i=0; i<n; i++)
   {
      printf("priority of p%d=", i);
      scanf("%d", &pt[i]);
   }
   for(i=0; i<n-1; i++)
   {
      for(j=0; j<n-i-1; j++)
      {
         if((pt[j]>pt[j+1]&&bt[j]<bt[j+1]))
         {
            temp1=bt[j];
            bt[j]=bt[j+1];
            bt[j+1]=temp1;
            temp2=pt[j];
            pt[j]=pt[j+1];
            pt[j+1]=temp2;
         }
      }
   }
   wt[i]=0;
   for(i=0; i<n; i++)
      wt[i]=wt[i-1]+bt[i-1];
   printf("waiting times are:\n");
   for(i=0; i<n; i++)
   {
      printf("%d\n", wt[i]);
   }
   for(i=0; i<n; i++)
   {
      wsum=wsum+wt[i];
      ta[i]=ta[i]+wt[i]+bt[i];
   }
   printf("total waiting time=%d\n", wsum);
   for(i=0; i<n; i++)
   {
      printf("turnaround time of %dth process=%d", i, ta[i]);
      printf("\n");
   }
   for(i=0; i<n; i++)
   {
      tsum=tsum+ta[i];
   }
   printf("total turnaround time is%d\n", tsum);
   wavg=(float)wsum/n;
   tavg=(float)tsum/n;
   printf("average waiting time=%. 2f\naverage turnaround time=%. 2f\n", wavg, tavg);
}
```

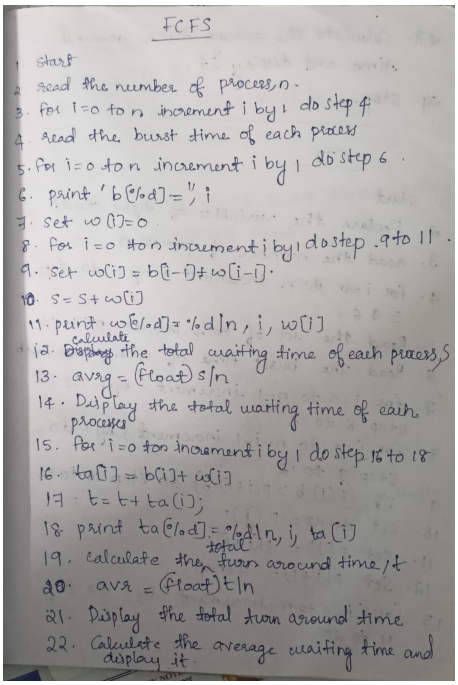
SJF

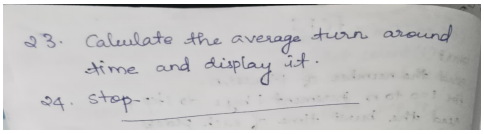
1. start
2. Declare the variables of type struct
3. read the number of process(n).
4. For i=0 to n increment i by 1 do step 5 & 6
5. Read the id of each process, p(i).id
6. Read the burst time of each process p(i).b
7. for i=0 to n-1 increment i by 1 do step 8 to 12
8. for j=0 to n-i-1 increment j by 1 do step 9 to 12
9. If (p[j].b > p(j+1).b) do step 10 to 12
10. set temp = p[j];
11. set p[j] = p(j+1);
12. Set p[j+1] = temp
13. for (i=0 to n increment i by 1 do step 14 to 25




```
#include <stdio.h>
typedef struct
{
   int id;
   int pr;
   int b;
}
 process;
void main()
{
   process p[10], temp;
   int i, j, n, w[50], t[50], c=0, d=0;
   float e, f;
   printf("enter the no of process:");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("enter the id of process:");
      scanf("%d", &p[i]. id);
      printf("enter the burst time:");
      scanf("%d", &p[i]. b);
   }
   for(i=0; i<n-1; i++)
   {
      for(j=0; j<n-i-1; j++)
      {
         if(p[i]. b>p[j+1]. b)
         {
            temp=p[j];
            p[j]=p[j+1];
            p[j+1]=temp;

         }
      }
   }
   printf("process\t burst time\t waiting time\t turn around time\t priority\n");
   for(i=0; i<n; i++)
   {
      if(i==0)
      {
         w[i]=0;
         t[i]=w[i]+p[i]. b;
         c=c+w[i];
         d=d+t[i];
         printf("p%d\t%d\t\t%d\t\t%d\t\t\n", p[i]. id, p[i]. b, w[i], t[i]);
         continue;
      }
      w[i]=w[i-1]+p[i-1]. b;
      t[i]=w[i]+p[i]. b;
      c=c+w[i];
      d=d+t[i];
      printf("p%d\t%d\t\t%d\t\t%d\t\t\n", p[i]. id, p[i]. b, w[i], t[i]);
   }
   printf("total wt=%d\n total ta=%d\n", c, d);
   e=(float)c/n;
   f=(float)d/n;
   printf("avg wt=%. 2f\n avg ta=%. 2f\n", e, f);

}
```

```c
#include <stdio.h>
void main()
{
 int n, p[20], b[20], w[20], ta[20];
 int i=0;
 int s=0;
 int t=0;
 float avrg=0;
 float avr=0;
 printf("Enter the number of process:");
 scanf("%d", &n);
 printf("Enter the burst time of processes:\n");
 for(i=0; i<n; i++)
 {
  printf("b[%d]=", i);
  scanf("%d", &b[i]);
 }
 w[i]=0;
 for(i=0; i<n; i++)
 {
  w[i]=b[i-1]+w[i-1];
  s=s+w[i];
  printf("w[%d]=%d\n", i, w[i]);
 }
 printf("total waiting time:%d\n", s);
 avrg=(float)s/n;
 printf("average waiting time:%f\n", avrg);
 for(i=0; i<n; i++)
 {
  ta[i]=b[i]+w[i];
  t=t+ta[i];
  printf("ta[%d]=%d\n", i, ta[i]);
 }
 printf("total turn around time:%d\n", t);
 avr=(float)t/n;
 printf("average turn around time:%f", avr);
}
```



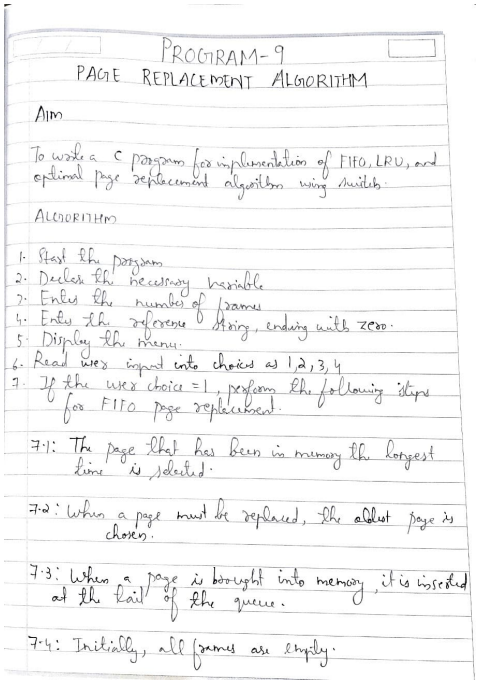


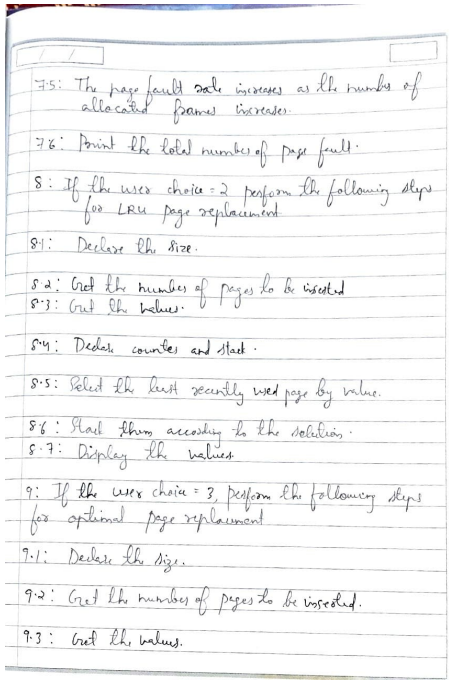

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#define SEGSIZE 100
int main(int argc, char *argv[])
{
    int shmid;
    key_t key;
    char *segptr;
    char buff[] = "poooda.....";
    key = ftok(".", 's');
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
    if ((shmid = shmget(key, SEGSIZE, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        if ((shmid = shmget(key, SEGSIZE, 0)) == -1) {
            perror("shmget");
            exit(1);
        }
    } else {
        printf("Creating a new shared memory segment\n");
        printf("SHMID: %d\n", shmid);
    }
    system("ipcs -m");
    if ((segptr = (char *)shmat(shmid, 0, 0)) == (char *)-1) {
        perror("shmat");
        exit(1);
    }
    printf("Writing data to shared memory...\n");
    strcpy(segptr, buff);
    printf("DONE\n");
    printf("Reading data from shared memory...\n");
    printf("DATA: %s\n", segptr);
    printf("DONE\n");
    if (shmdt(segptr) == -1) {
        perror("shmdt");
    }
    printf("Removing shared memory segment...\n");
    if (shmctl(shmid, IPC_RMID, 0) == -1)
        printf("Can't remove shared memory segment...\n");
    else
        printf("Removed successfully\n");
    return 0;
}
```

Algorithm for Shared Memory Communication
Step 1: Start the program.
Step 2: Declare required variables:
shmid → Shared memory ID
key → Key for shared memory
segptr → Pointer to shared memory
buff[] → Data to write in shared memory
SEGSIZE → Size of shared memory (100 bytes)
Step 3: Generate a unique key using ftok().
Step 4: Create or access shared memory using shmget():
If shared memory already exists → access it.
Else → create a new shared memory segment.
Step 5: If shared memory created newly, display SHMID and message.
Step 6: Display shared memory details using system("ipcs -m").
Step 7: Attach shared memory segment to process address space using shmat().
Step 8: Write data (buff) to shared memory using strcpy().
Step 9: Display message confirming data written.
Step 10: Read and display the data from shared memory.
Step 11: Remove the shared memory segment using shmctl() with IPC_RMID.
Step 12: Display appropriate message based on removal success or failure
Step 13: End the program.

```c
memory management scheme
#include <stdio.h>
void firstFit(int mem[], int n, int process[], int m, int
allocation[]) {
for(int i = 0; i < m; i++) {
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
allocation[i] = j;
mem[j] -= process[i];
break;}}}}
void bestFit(int mem[], int n, int process[], int m,
int allocation[]) {
for(int i = 0; i < m; i++) {
int bestIdx = -1;
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
if(bestIdx == -1 || mem[j] < mem[bestIdx]) {
bestIdx = j;}}}
if(bestIdx != -1) {
allocation[i] = bestIdx;
mem[bestIdx] -= process[i];}}}
void worstFit(int mem[], int n, int process[], int m,
int allocation[]) {
for(int i = 0; i < m; i++) {
int worstIdx = -1;
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
if(worstIdx == -1 || mem[j] > mem[worstIdx]) {
worstIdx = j;}}}
if(worstIdx != -1) {
allocation[i] = worstIdx;
mem[worstIdx] -= process[i];}}}
int main() {
int n, m, choice;
int mem[10], process[10], temp_mem[10],
allocation[10];
printf("Enter number of memory blocks: ");
scanf("%d", &n);
printf("Enter size of each block:\n");
for(int i = 0; i < n; i++) {
scanf("%d", &mem[i]);}
printf("Enter number of processes: ");
scanf("%d", &m);
printf("Enter size of each process:\n");
for(int i = 0; i < m; i++) {
scanf("%d", &process[i]);
allocation[i] = -1;}
printf("\nMemory Allocation Algorithms:\n");
printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n");
printf("Enter choice: ");
scanf("%d", &choice);
for(int i = 0; i < n; i++) {
temp_mem[i] = mem[i];}
switch(choice) {
case 1:
firstFit(temp_mem, n, process, m, allocation);
break;
case 2:
bestFit(temp_mem, n, process, m, allocation);
break;
case 3:
worstFit(temp_mem, n, process, m, allocation);
break;
default:
printf("Invalid choice!\n");
return 1;}
printf("\nProcess\tSize\tBlock Allocated\n");
for(int i = 0; i < m; i++) {
printf("%d\t%d\t", i+1, process[i]);
if(allocation[i] != -1) {
printf("%d\n", allocation[i] + 1);
} else {
printf("Not Allocated\n");}}
printf("\nRemaining Memory Blocks:\n");
for(int i = 0; i < n; i++) {
printf("Block %d: %d\n", i+1, temp_mem[i]);
}
return 0;
}
```

```c
page replacement
#include <stdio.h>
#include <stdbool.h>
// Function prototypes
int fifo(int pages[], int n, int frames);
int lru(int pages[], int n, int frames);
int optimal(int pages[], int n, int frames);
int main() {
int choice, frames, n;
printf("Enter number of frames: ");
scanf("%d", &frames);
printf("Enter number of page references: ");
scanf("%d", &n);
int pages[n];
printf("Enter page reference string: ");
for(int i = 0; i < n; i++) {
scanf("%d", &pages[i]);}
printf("\nPage Replacement Algorithms:\n");
printf("1. FIFO\n2. LRU\n3. Optimal\n");
printf("Enter your choice: ");
scanf("%d", &choice);
int faults = 0;
switch(choice) {
case 1:
faults = fifo(pages, n, frames);
break;
case 2:
faults = lru(pages, n, frames);
break;
case 3:
faults = optimal(pages, n, frames);
break;
default:
printf("Invalid choice!\n");
return 1;}
printf("\nTotal Page Faults: %d\n", faults);
return 0;}
// FIFO Implementation
int fifo(int pages[], int n, int frames) {
int mem[frames], queue[frames];
int front = 0, faults = 0;
for(int i = 0; i < frames; i++) mem[i] = -1;
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
break;}}
if(!found) {
mem[front] = pages[i];
front = (front + 1) % frames;
faults++;}}
return faults;}
// LRU Implementation
int lru(int pages[], int n, int frames) {
int mem[frames], counter[frames];
int faults = 0, time = 0;
for(int i = 0; i < frames; i++) {
mem[i] = -1;
counter[i] = 0;}
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
counter[j] = ++time;
break;}}
if(!found) {
int lru = 0;
for(int j = 1; j < frames; j++) {
if(counter[j] < counter[lru]) lru = j;}
mem[lru] = pages[i];
counter[lru] = ++time;
faults++;}}
return faults;}
// Optimal Implementation
int optimal(int pages[], int n, int frames) {
int mem[frames], faults = 0;
for(int i = 0; i < frames; i++) mem[i] = -1;
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
break;}}
if(!found) {
faults++;
int replace = -1, farthest = i;
for(int j = 0; j < frames; j++) {
int k;
for(k = i + 1; k < n; k++) {
if(mem[j] == pages[k]) break;}
if(k > farthest) {
farthest = k;
replace = j;}}
if(replace == -1) replace = 0;
mem[replace] = pages[i];}}
return faults;
}
```





9.4: Check future page references to determine which page will not be needed for longest time

9.5: Select that page for replacement.

9.6: Replace the selected page with new page.

9.7: Display the values.

10: If the user choice = 4, exit the program.

11: Stop the program

PROGRAM - 8

MEMORY ALLOCATION METHODS FOR FIXED PARTITION

Aim

To write a program to implement memory management scheme First fit, Best fit and Worst fit.

ALGORITHM

Step1: Start

Step2: Read number of memory blocks, m

Step3: Read size of memory blocks

Step4: Read number of processes, n

Step5: Read size of processes.

Step6: For each process i from 1 to n, set allocation[i] = -1

Step7: Display choice
1 → First fit
2 → Best fit
3 → Worst fit

Step8: Read choice.

Step9: For each block i from 1 to n, copy mem[i] to temp_mem[i]

Step10: if choice == 1 (First fit)
for each process i from 1 to n:
for each block j from 1 to m
if temp_mem[j] is large enough for process[i]
Assign allocation[i] = j
Reduce temp_mem[j] by process[i]
Break inner loop

Step11: else if choice == 2 (Best fit)
for each process i from 1 to n
Set bestIdx = -1
for each block j from 1 to m:
if temp_mem[j] fits process[i] and is smaller than the current bestIdx block:
Set bestIdx = j
if bestIdx is found, assign allocation[i] = bestIdx
Reduce temp_mem[bestIdx] by process[i]

Step12: else if choice == 3 (worst fit)
For each process i from 1 to n:
Set worstIdx = -1
for each block j from 1 to m:
if temp_mem[j] fits process[i] and is larger than the current worstIdx block, Set worstIdx = j
if worstIdx is found, assign allocation[i] = worstIdx
Reduce temp_mem[worstIdx] by process[i].

Step13: else, print "Invalid choice!" and exit.

Step14: Print Process, Size, Block Allocated for each process i from 1 to n:
- if allocation[i] is assigned, print the block number
- else, print "Not Allocated".

Step15: Print updated sizes of all memory blocks: temp_mem[1] to temp_mem[m].

Step16: Stop