L SAI KISHORE
36110657

EXP NO:-2 DATE:-

EXP NAME:-SEARCHING A SUBSTRING IN A GIVEN TEXT

AIM:-

To write a program for searching a substring in a given text by using shell programming.

ALGORITHM:-

1) To select a substring from string using ${string: starting position :root position}
2) Comparing two strings is done by {$s1=$s2}
3) To check for zero length string use [-z String]
4) To check for empty string use [String]
5) To check for non zero length string use –n as [-n $string]
6) The length of string is obtained by ${#string}

PROGRAM:-

```
S1=”this is a text”

S2=${s1:6:6}

echo $2

a=”good”

b=”bad”

if[$a=$b]

then

echo “a is equal to b”

else

echo “a is not equal to b”
```

```
fi
a=" "
if[-z $a]
then
echo "string length is zero"
else
echo "string length is non zero"
fi
if[$a]
then
echo "string is not empty"
else
echo "string is empty"
fi
if[-n $a]
then
echo "string is not zero"
else
echo "string length is zero"
fiif[$a!=$b]
then
echo "a is not equal to b"
else
echo "a is equal to b"
fi
s="sample string"
```

```
echo ${#s}
```

OUTPUT:-

```
a is not equal to b

string length is not zero

string is not empty

string length is not zero
a is not equal to b
```

RESULT:-

EXP NO:-3 DATE:-

EXP NAME:-MENU BASED MATH CALCULATOR

AIM:-

To write a program for menu based math calculator using shell scripting commands.

ALGORITHM:-

1) Read the operator
2) Read the operands
3) Using the operator as choice for switch case write case for operators
4) End switch case
5) Stop

PROGRAM:-

```
echo "1.Addition 2.Subtraction 3.Multiplication 4.Division"

read n

echo "Enter the operends"

read a

read b

case $n in

"1") echo "$a +$b =`expr $a \ + $b`";

"2") echo "$a-$b=`expr $a  \ - $b`";

"3") echo "$a*$b=`expr $a \ * $b`";

"4") echo "$a/$b=`expr $a \ / $b`";

Esac
```

OUTPUT:-

```
Enter the operator
```

```
/

Enter the operands

10

2

10/5=2
```

RESULT:-

EXP NO:-4 **DATE:-**

EXP NAME:-PRINTING PATTERN USING LOOP STATEMENT

AIM:-

To print a pattern using loop statement by using shell scripting commands.

ALGORITHM:-

1) Read the number given.
2) Initialize the for loop where i<=$n.
3) Initiallize one more loop inside the above loop with j<=$i.
4) Print "*" and close the two loops.
5) Continue until the required root loops(rows) reached.

PROGRAM:-

```
echo "enter a number"

read n

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

do

for((j=1;j<=Si;j++))

do

echo –n "*"

done

echo " "

done
```

OUTPUT:-

```
Enter a number

3

*

**

***
```

RESULT:-

EXP NO:-5 **DATE:-**

EXP NAME:-CONVERTING FILES NAMES FROM UPPERCASE TO LOWERCASE

AIM:-

To write a program for converting files from uppercase case to lowercase.

ALGORITHM:-

1) Get the file name.
2) store the name in a variable.
3) Apply conversion to that variable.
4) Store it in other variable.
5) Finally display the converted file name.

PROGRAM:-

```
for i in vishnu

do

echo “before conversion is”

echo $i;

j=`echo $i|tr’[a-z]’’[A-Z]’`

echo “after conversion is”

echo $j;

done
```

OUTPUT:-

```
Before conversion is

vishnu

After conversion is

VISHNU
```

RESULT:-

EXP NO:-6 **DATE:-**

EXP NAME:-MANIPULATING DATE/TIME/CALENDAR

AIM:-

To write a program for manipulating the date/time/calendar using shell commands

ALGORITHM:-

1) To display login name logname variable is used.
2) Display userinfo using variable who I am.
3) Display date(present) using variable date.
4) Display current directory using pwd variable.

PROGRAM:-

```
echo “hello,$LOGNAME”

echo “user is,’whoi am’”

echo “date is,’date’”

echo “current directory,$(pwd)”
```

OUTPUT:-

```
hello,admin

user is ,cse pts/12 2017-02-16 13:48(scse008.mobilelab.com)

date is, 2017-02-16 13:48

Current directory,/home/cse
```

RESULT:-

EXP NO:-7 **DATE:-**

EXP NAME:-SHOWING VARIOUS SYSTEM INFORMATION

AIM:-

To write a program in vi editor to show various information using shell command

ALGORITHM:-

1. Get the system information such as network name and node name, kernel name, kernel version e.t.c .

2 Network $node name=$(uname –n).

Kernel name=$(uname –s)

Kernel ru=$(uname –a).

Operating system =$(uname –m).

All information $(uname –A).

PROGRAM:-

```
echo "NETWORK =  $(uname –n)"
echo "Kernel NAME =  $(uname –s)"
echo "Kernel version = $(uname –v)"
echo "Operating System =  $(uname –m)"
echo "All information = $(uname –a)"
```

OUTPUT:-

```
NETWORK=mobility

Kernel NAME=Linux

Kernel version =#1 SMP Thu May 6 18:27:11 UTC 2010

Operating system = i686

All information = Linux mobility 2.6.33.3-85.fc13.i686.PAE #1 SMP Thu May 6
18:27:11 UTC 2010 i686 i686 i386 GNU/Linux
```

RESULT:

L SAI KISHORE
36110657

EXP NO: 8 **DATE:**

EXP NAME:IMPLEMENTATION OF PROCESS SCHEDULING MECHANISM – FCFS, SJF, PRIORITY QUEUE.

AIM:

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

8.(A) FCFS SCHEDULING:

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

(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)

Step 6: Calculate

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

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

Step 7: Stop the process

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
Int main()
{
Int n,bt[20],wt[20],tat[20],avwt=0;avtat=0;t,j;
Clrscr();
printf(“enter the total no of processes:”);
scanf(“%d”,& n);
printf(“\n enter process burst time\n”);
for(i=0;i<n;i++)
{
print(“p[%d]”,i+1);
scanf(“%d”,& bt[i]);
}
wt[0]=0;
for(i=0;i<n;i++)
{
wt[i]=0;
for(j=0;j<n;j++)
wt[i]+=bt[j];
}
Printf(“\n process\t\t burst time\t waiting time\t turnaround time”);
for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];
avwt+=wt[i];
avtat+=tat[i];
printf(“\n p[%d]\t\t%d\t\t%d\t\t%d”,i+1,bt[i],wt[i],tat[i]);
}
avwt/=I;
avtat/=I;
printf(“\n average waiting time:%d”,avwt);
printf(“\n average turnaround time:%d”,tat);
getch();
return 0;
}
```

OUTPUT:

```
Enter totalnumber of processes:2
Enter process burst  time
p[1]:42
p[2]: 95


Process       Burst time       Waiting time      Turnaround time
p[1]          42               0                 42
p[2]          95               42                137

Average waiting time: 21

Average turnaround time: 89
```

8. (B) SJF

ALGORITHM FOR SJF:

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: 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

(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)

Step 7: Calculate

(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

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];
float tot,t;
```

```
printf("Enter the no.of process");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("Enter process id and time");
scanf("%s%d",s[i],&a[i]);
}
wt[0]=0;
a[0]=0;
t=tot=0;
for(i=1;i<=n;i++)
{
   for(j=i+1;j<=n;j++)
   {
      if(a[i]>a[j])
      {
         temp=a[i];
         a[i]=a[j];
         a[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”);
for(i=1;i<=n;i++)
{
wt[i]=wt[i-1]+a[i-1];
trn[i]=wt[i]+a[i];
printf("%s %d %d \n",s[i],wt[i],trn[i]);
tot=tot+wt[i];
t=t+trn[i];
}
```

```
printf("Average waiting time=%f Average turn around time=%f",tot/n,t/n);
getch();
}
```

OUTPUT:

```
Enter number of process:3
Enter burst time:
P1:27
P2:28
P3:22

Process      burst time      waiting time      turnaround time
P3           22              0                 22
P1           27              22                49
P2           28              49                77
Average waiting time=23.666666
Average turnaround time=49.333332
```

8. (C).PRIORITY SCHEDULING.

ALGORITHM FOR PRIORITY 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: 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

(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)

Step 7: Calculate

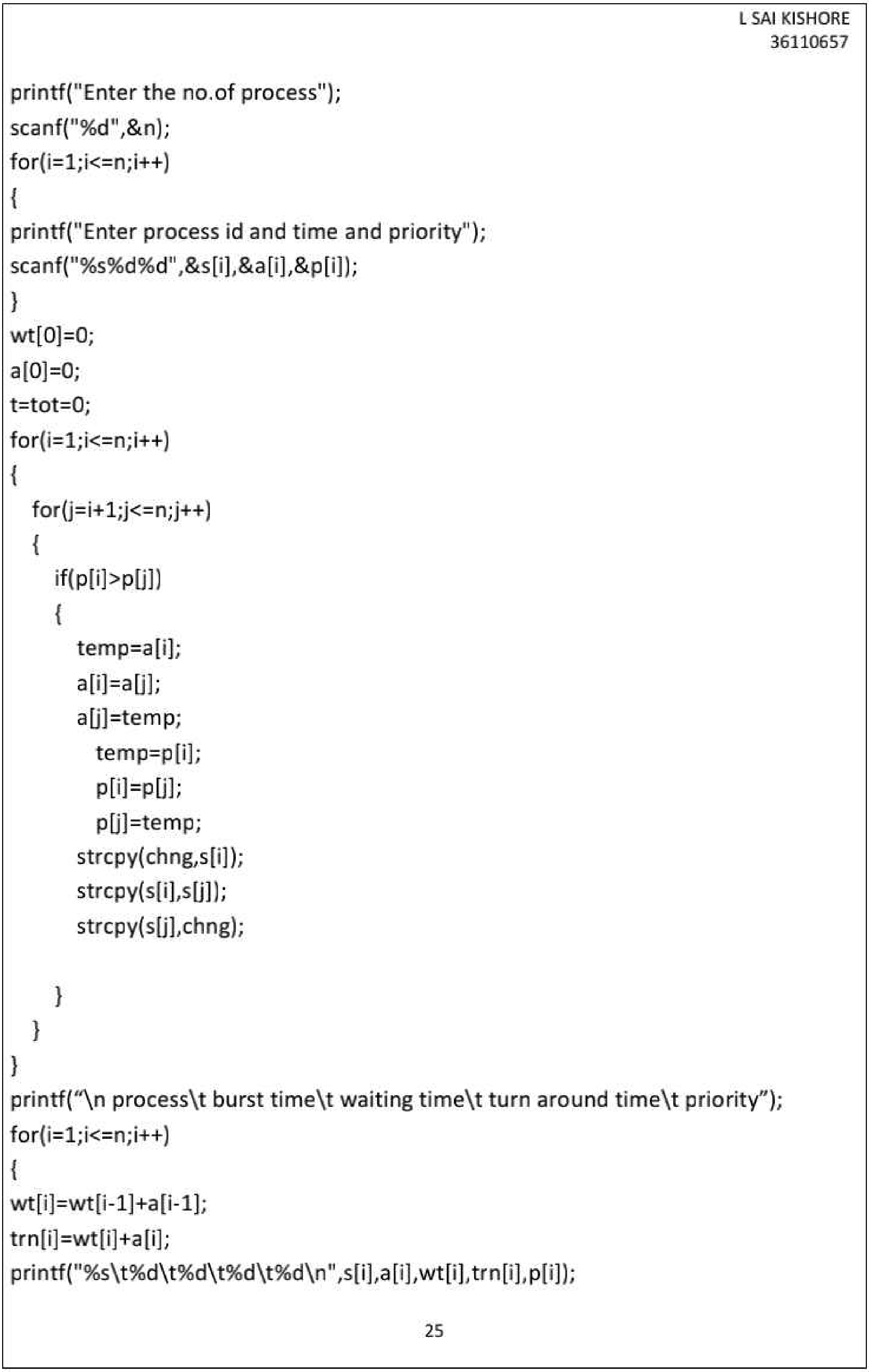
(a) Average waiting time = Total waiting Time / Number of process

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

Step 8: Stop the process

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;
clrscr();
```



```
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();
}
```

OUTPUT:

```
Enter no of processes:2
Enter the process is and time and priority :
34
65
26
Enter the process is and time and priority :
3
55
87

Process    Burst time   Waiting time   Turnaround time   Priority
34            65             0              65              26
3             55             65             120             87
Average waiting time=032.500000 Average turnaround time=092.500000
```

RESULT:

EXP NO:9 **DATE:**

EXP NAME:READER-WRITERS PROBLEM

AIM:

To write a program to implement readers and writers problem

ALGORITHM:

```
Start ;
/* Initialize semaphore variables*/
integer mutex=1;          // Controls access to RC
integer DB=1;             // controls access to data base
integer RC=0;             // Number of process reading the database currently
1.Reader( )      // The algorithm for readers process
Repeat continuously
        DOWN(mutex);          // Lock the counter RC
        RC=RC+1;              // one more reader
If(RC=1)DOWN(DB);    // This is the first reader.Lock the database for reading
UP(mutex);           // Release exclusive access to RC
Read database();     // Read the database
DOWN(mutex);         // Lock the counter RC
RC=RC-1;             // Reader count less by one now
If(RC=0)UP(DB);      // This is the last reader .Unlock the database.
UP(mutex);           // Release exclusive access to RC
End
2.Writer( )    // The algorithm for Writers process
Reepeat continuously
DOWN(DB);          // Lock the database
Write_Database(); // Read the database
UP(DB);            // Release exclusive access to the database
End
```

Step a: initialize two semaphore mutex=1 and db=1 and rc,(Mutex controls the access to read count rc)

Step b: create two threads one as Reader() another as Writer()

Reader Process:

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)

PROGRAM:

Cv

```
#include<stdio.h>

int x=1,rc=0,readcount=1;

void p 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*a)
{
*d=*d+1;
}
void reader()
{
int flag=1;
while(flag==1)
{
p(&reader count);
rc=rc+1;
if(rc==1)
p1(&x)
v(&read count);
printf("\n reader is reading");
p(&read count);
rc=rc-1;
if(rc==0)
v1(&x);
v(&read count);
flag=0;
}
}
void writer()
{
p1(&x);
printf("\n writer is writing");
v1(&x);
}
void main()
{
reader();
writer();
reader();
writer();
}
```

OUTPUT:

```
Reader is reading
Writer is writing
Reader is reading
Reader is reading
Writing is writing
```

RESULT:

L SAI KISHORE
36110657

EXP NO: 10 **DATE:**

EXP NAME: DINING PHILOSOPHERS PROBLEM

AIM:

Write a program to solve the Dining Philosophers problem.

ALGORITHM:

1. Initialize the state array S as 0, Si =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 philosopher 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)

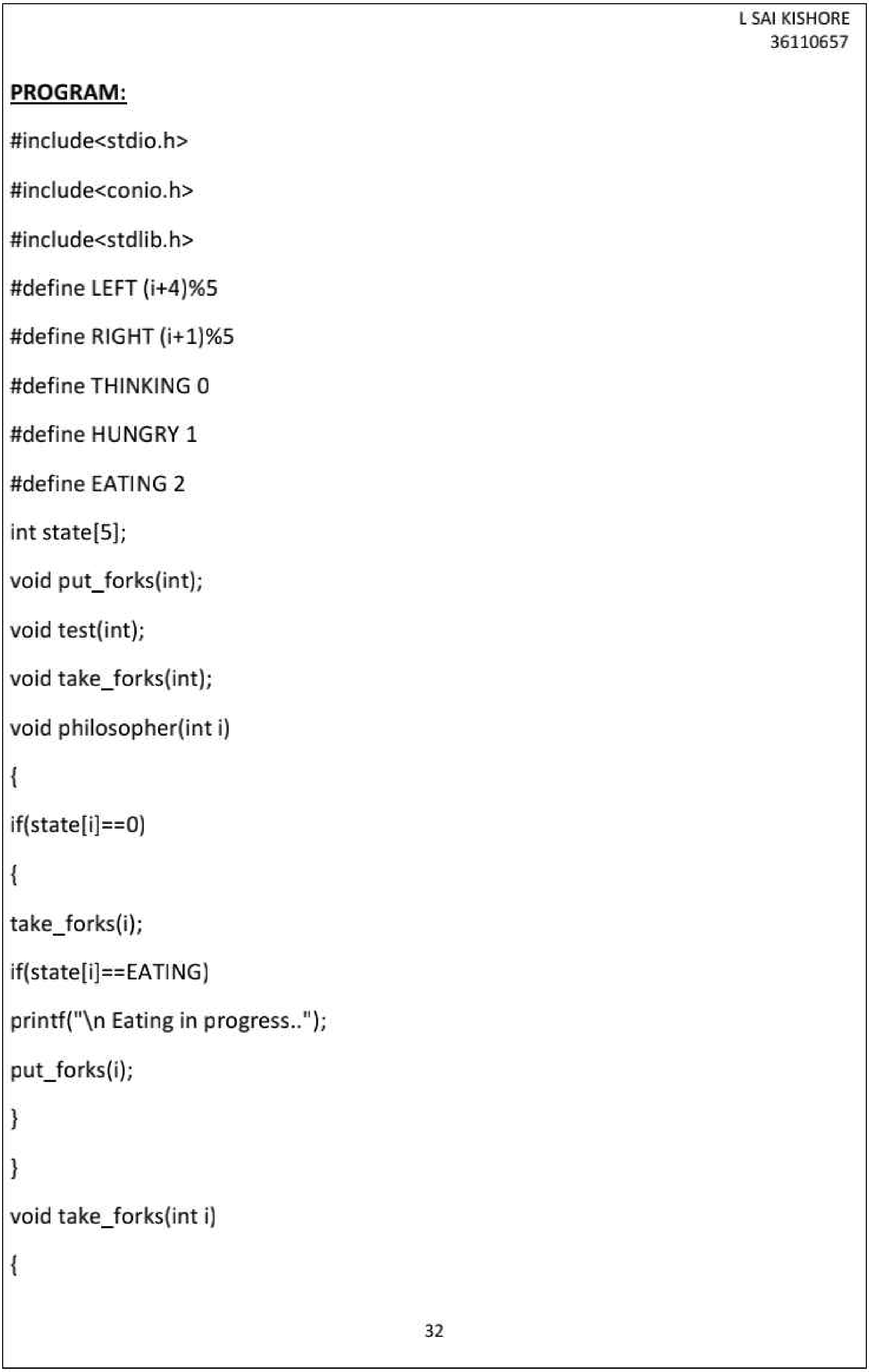
Step 1: set S[i]=0 I.e. the philosopher i is thinking

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.



PROGRAM:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define LEFT (i+4)%5

#define RIGHT (i+1)%5

#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 progress..");

put_forks(i);

}

}

void take_forks(int i)

{
```

```
state[i]=HUNGRY;

test(i);

}

void put_forks(int i)

{

state[i]=THINKING;

printf("\nphilosopher %d has completed its work",i);

test(LEFT);

test(RIGHT);

}

void test(int i)

{

if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)

{

printf("\nphilosopher %d can eat",i);

state[i]=EATING;

}

}

void main()

{

int i;

for(i=1;i<=5;i++)

state[i]=0;

printf("\n\n\t\t\t DINING PHILOSOPHERS PROBLEM");

printf("\n\t\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
```

```
printf("\n ALL THE PHILOSOPHERS ARE THINKING !!....\n",i);
for(i=1;i<=5;i++)
{
printf("\n\n the philosophers %d falls hungry\n",i);
philosopher(i);
getch();
}
}
```

OUTPUT:

```
DINNER'S PHILOSOPHERS PROBLEM
 ALL THE PHILOSOPHERS ARE THINKING !!....
The philosophers 1 falls hungry
Philosopher 1 can eat
Philosopher 1 has completed its work
The philosophers 2 falls hungry
Philosopher 2 can eat
Philosopher 2 has completed its work
The philosophers 3 falls hungry
Philosopher 3 can eat
Philosopher 3 has completed its work
The philosopher 4 falls hungry
Philosopher 4 can eat
Philosopher 4 has completed its work
The philosophers 5 falls hungry
```

```
Philosopher 5 can Philosopher 5 has completed its word
```

RESULT:

L SAI KISHORE
36110657

EXP NO: 11 DATE:

EXP NAME: FIRST FIT , WORST FIT, BEST FIT ALLOCATION STRATEGY

AIM:

To implement

a) First fit

b) Best fit

c) Worst fit &

d) To make comparative study

THEORY:

Memory Management Algorithm

In an environment that supports dynamic memory allocation, a number of strategies are used to allocate a memory space of size n (unused memory partition) from the list free holes to the processes that are competing for memory.

First Fit: Allocation the first hole which is big enough.

Best Fit: Allocation the smallest hole which is big enough

Worst Fit: Allocation the largest hole which is big enough

ALGORITHM:

Step 1: Start the program.

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.

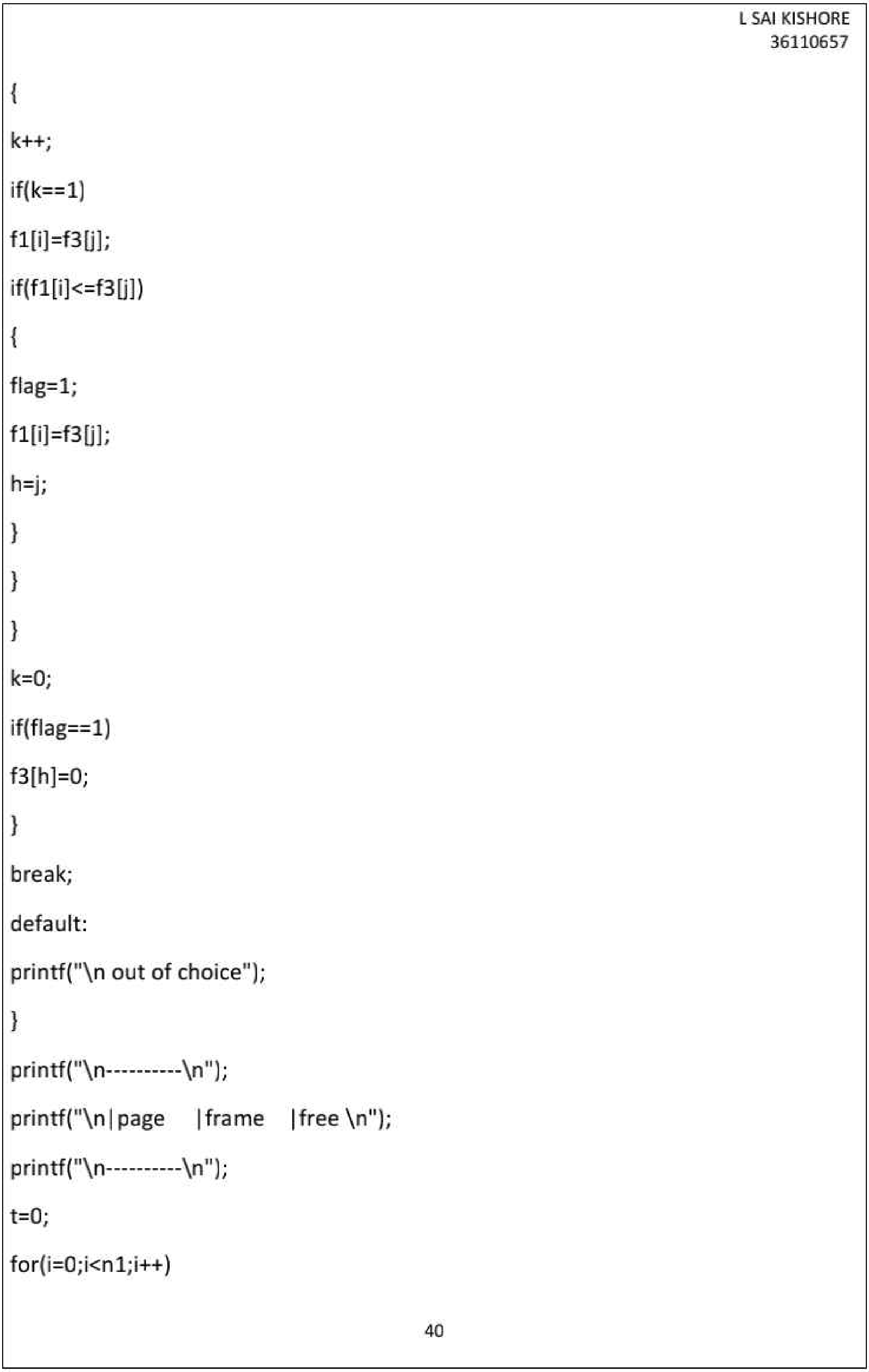
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>

#include<conio.h>

int main()

{

int p[20],f[20],min,minindex,n,i,j,c,f1[20],f2[20],f3[20],k=0,h=0,flag,t=0,n1;

clrscr();

printf("enter the number of memory partitions:\n");

scanf("%d",&n);

printf("enter the number of process");

scanf("%d",&n1);

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

{

printf("\n enter the memory partition size %d:",i+1);

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

f2[i]=f[i];

f3[i]=f[i];

}

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

{

printf("\n enter the page size %d:",i+1);
```

```
scanf("%d",&p[i]);
}
do
{
printf("\n1.first fit\n");
printf("\n2.best fit\n");
printf("\n3.worst fit\n");
printf("\nenter your choice\n");
scanf("%d",&c);
switch(c)
{
case 1:
for(i=0;i<n1;i++)
{
for(j=0;j<n;j++)
{
f1[i]=0;
if(p[i]<=f[j])
{
f1[i]=f[j];
f[j]=0;
break;
}
}
}
```

```
break;
case 2:
for(i=0;i<n1;i++)
{
min=9999;
minindex=-1;
for(j=0;j<n;j++)
{
if(p[i]<=f2[j] && f2[j]!=0 && min>f2[j])
{
min=f2[j];
minindex=j;
}
}
f1[i]=f[minindex];
f2[minindex]=0;
}
break;
case 3:
for(i=0;i<n1;i++)
{
f1[i]=0;
for(j=0;j<n;j++)
{
if(p[i]<f3[j])
```



```
{
k++;
if(k==1)
f1[i]=f3[j];
if(f1[i]<=f3[j])
{
flag=1;
f1[i]=f3[j];
h=j;
}
}
}
k=0;
if(flag==1)
f3[h]=0;
}
break;
default:
printf("\n out of choice");
}
printf("\n----------\n");
printf("\n|page    |frame    |free \n");
printf("\n----------\n");
t=0;
for(i=0;i<n1;i++)
```

```
{
h=f1[i]-p[i];
if(h<0)
h=0;
printf("\n%d\t\t%d\t\t%d",p[i],f1[i],h);
t=t+h;
}
printf("\n----------\n");
printf("\n total free spae in memory:%d",t);
}
while(c<4);
}
```

OUTPUT:

```
Enter the number of memory partitions:
3
Enter the number of process:  2
Enter the memory partition size 1: 242
Enter the memory partition size 2: 200
Enter the memory partition size 3: 350
Enter the page size 1: 100
Enter the page size 2: 300
Enter the page size 3: 150
1. first fit
```

2. best fit

3. worst fit

Enter your choice: 1

Page	frame	Free
100	242	142
300	0	0

Total free space in memory : 142

1. first fit
2. best fit
3. worst fit

Enter your choice: 2

Page	frame	Free
100	200	100
300	0	0

Total free space in memory: 100

1. first fit

2. best fit

3. worst fit

Enter your choice: 3

Page	frame	Free
100	250	150
300	0	0

Total free space in memory: 150

1. first fit
2. best fit
3. worst fit

```
Enter your choice: 4
```

RESULT:

EXPNO: 12 **DATE:**

EXP NAME: BANKERS ALGORITHM

AIM:

Write a program to implement Banker's Algorithm

ALGORITHM:

This algorithm was suggested by Dijkstar, the name banker is used here to indicate that it uses a banker's activity for providing loans and receiving payment against the given loan. This algorithm places very few restrictions on the processes competing for resources. Every request for the resource made by a process is thoroughly analyzed to check, whether it may lead to a deadlock situation. If the result is yes then the process is blocked on this request. At some future time, its request is considered once again for resource allocation. So this indicated that, the processes are free to request for the allocation, as well as de-allocation of resources without any constraints. So this generally reduces the idling of resources.

Suppose there are (P) number of Processes and (r) number of resources then its time complexity is proportional to P x r2

At any given stage the OS imposes certain constraints on any process trying to use the resource. At a given moment during the operation of the system, processes P, would have been allocated some resources. Let these allocations total up to S.

Let (K=r-1) be the number of remaining resources available with the system. Then k>=0 is true, when allocation is considered.

Let maxk be the maximum resource requirement of a given process Pi.

Actk be the actual resource allocation to Pi at any given moment.

Then we have the following condition.

Maxk<=p for all k and

To

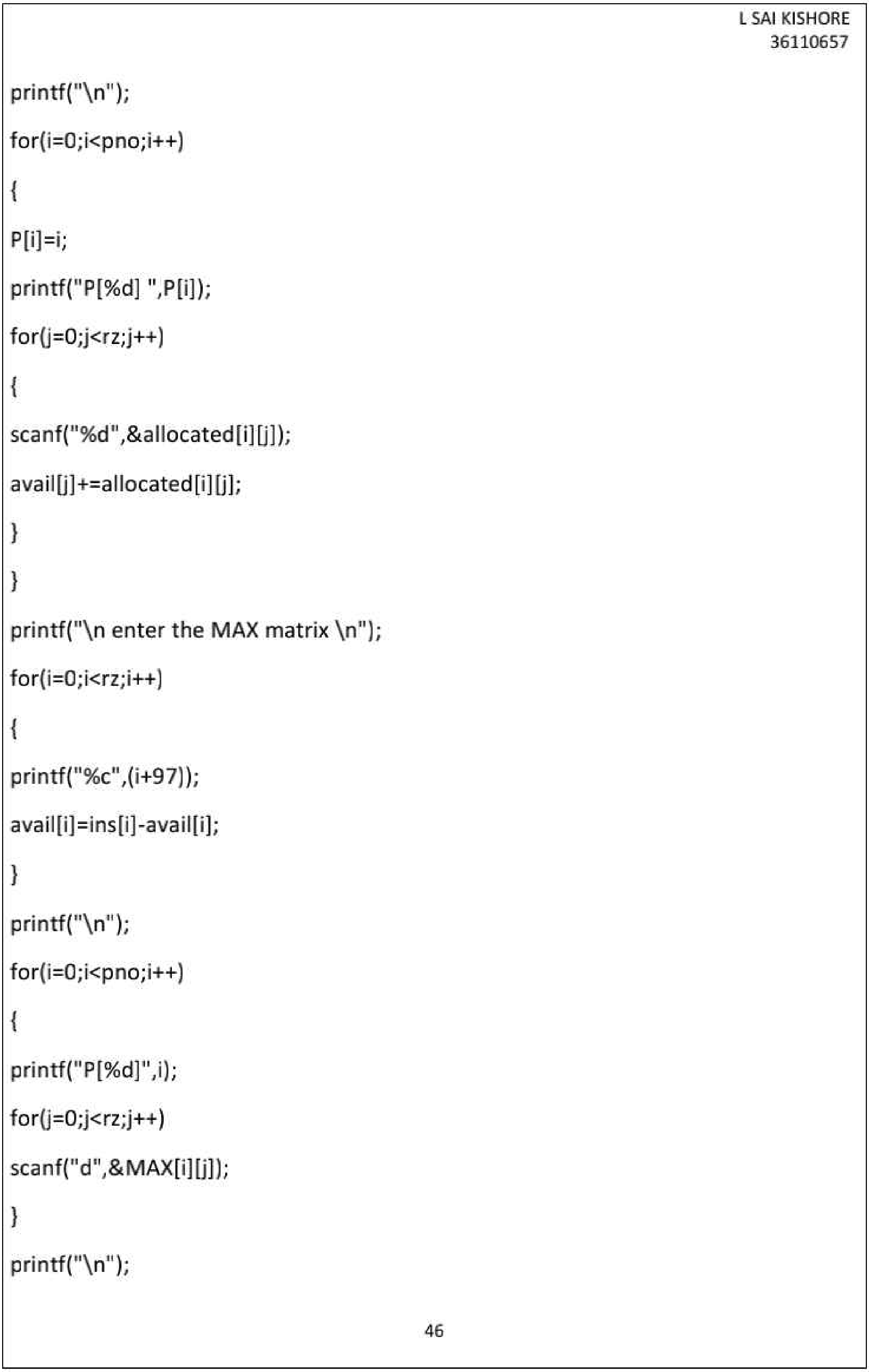
Disadvantages of Banker's algorithm:

1. The maximum number of resources needed by the processes must be known in advance

2. The no of processes should be fixed.

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;

clrscr();

printf("\n enter the number of resources:");

scanf("%d",&rz);

printf("\n enter 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("\n enter the number of processes:");

scanf("%d",&pno);

printf("\n enter 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("\n enter 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();}
```

OUTPUT:

```
Enter the number of resources : 2

Enter the max instances of each resources

a=10

b=6

Enter the number of processes = 2

Enter the allocation matrix

a  b

P[0] 4 6

P[1] 5 8

Enter the max matrix

a  b

P[0] 3 7

P[1] 8 6
```

RESULT:

L SAI KISHORE
36110657

EXP NO:13 **DATE:**

EXP NAME: IMPLEMENT THE PRODUCER CONSUMER PROBLEM USING SEMAPHORE

AIM:

To write a program to implement producer consumer problem using semaphore.

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 consumer. If yes check whether the buffer is empty

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

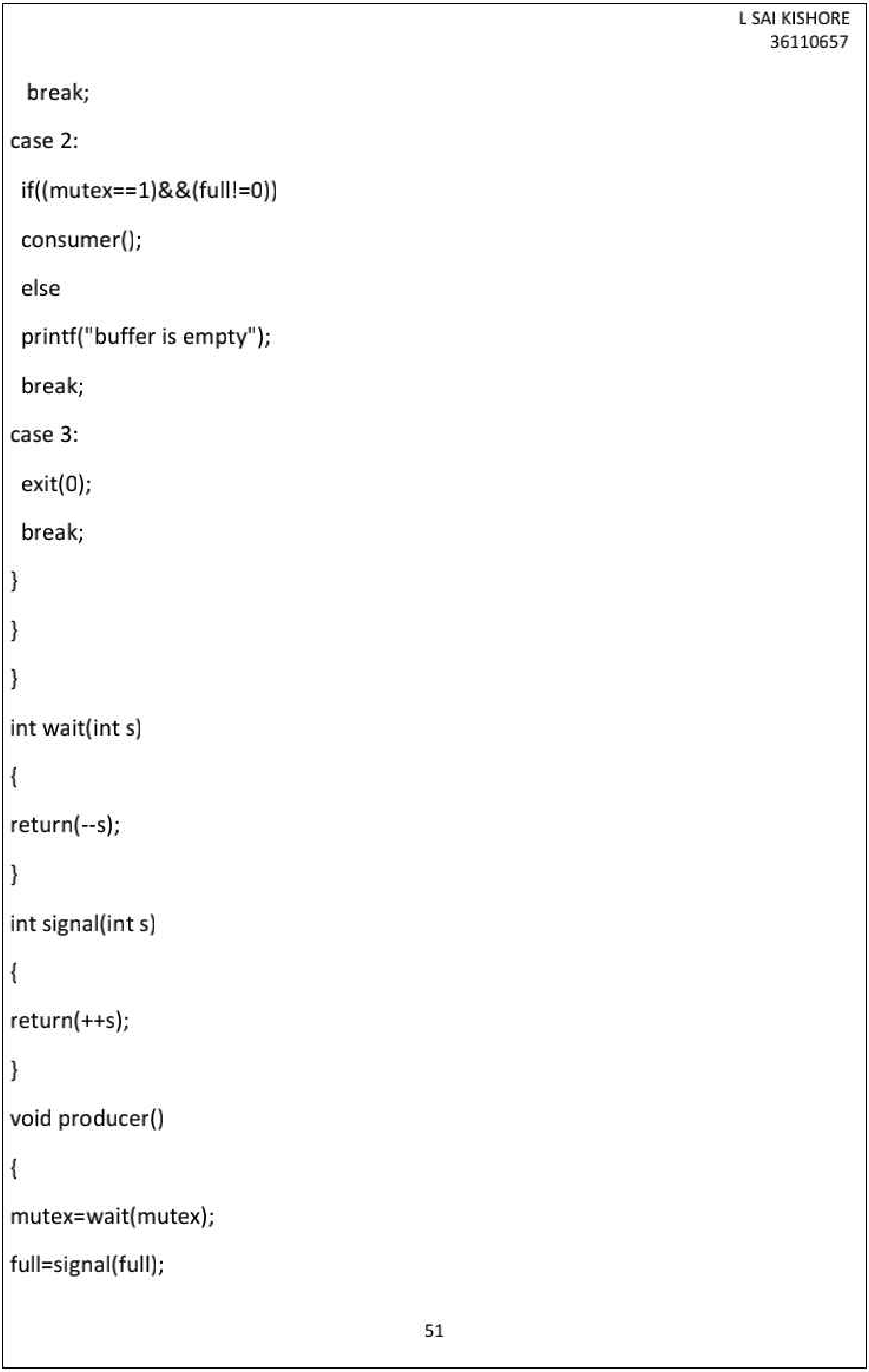
Step 9: If the buffer is empty, the consumer has to wait.

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

Step 11: Terminate the process.

PROGRAM:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

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 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);
 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 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:

```
   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 : 1
```

```
Buffer is full

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
```

RESULT:

EXP NO: 14 DATE:

EXP NAME: TO IMPLEMENT THE MEMORY MANAGEMENT POLICY-PAGING

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>

main()

{

int size,m,n,pgno,pagetable[3]={5,6,7},i,j,frameno;

double m1;

int ra=0,ofs;

clrscr();
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:")]

prscanf("%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();

}
```

OUTPUT:

```
Enter process size: 412

Total no. of pages: 3

Enter relative address: 1

2643

Page no=2

Page table

0[5]

1[6]

2[7]
```

RESULT: