OPERATING SYSTEM

LAB MANUAL

Subject Code:PCCS7304
Class:3rd Year (5th / 6th Semester) CSE

Prepared By
Mr. R. P. Nayak
Assistant Professor

GOVT.COLLEGE OF ENGINEERING,KALAHANDI
BHAWANIPATNA

Department of Computer Science & Engineering
Government College of Engineering, Kalahandi,
Bhawanipatna 766002, Odisha

Government College of Engineering, Kalahandi, Bhawanipatna 766002, Odisha

Program Outcomes	
PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

OPERATING SYSTEM LABORATORY SYLLABUS

1. Basic UNIX Commands.
2. Linux Administrative commands.
3. UNIX Shell Programming.
4. Programs on process creation and synchronization, inter process communication including shared memory, pipes and messages. (DinningPhilosopher problem / Cigarette Smoker problem / Sleeping barber problem)
5. Programs on UNIX System calls.
6. Simulation of CPU Scheduling Algorithms. (FCFS, RR, SJF, Priority, Multilevel Queuing)
7. Simulation of Banker's Algorithm for Deadlock Avoidance, Prevention
8. Program for FIFO, LRU, and OPTIMAL page replacement algorithm.

PCCS7304 OPERATING SYSTEM LAB (0-0-3)

OBJECTIVES:

The objective of this lab is to:

- Use basic unix commands.
- Learn shell programming.
- **Analyze** and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, Priority, and Multilevel Queuing.
- Programs on UNIX System calls.
- Be exposed to process creation and inter process communication.
- Implement memory management schemes and page replacement schemes.

LIST OF EXPERIMENTS:

1. Basic UNIX Commands & First C Program in Unix Environment.
2. UNIX Shell Programming
3. Implement the following CPU scheduling algorithms:
   a) FCFS b) SJF c) Round Robin d) Priority
4. Programs on UNIX System calls.
5. Program to simulate the concept of Dining-Philosophers problem.
6. Implement Bankers Algorithm for Dead Lock Avoidance.
7. Implement the following page replacement algorithms:
   a) FIFO b) LRU c) OPTIMAL Page Replacement

OUTCOMES:

At the end of the course, the student should be able to:

- Get aquanted with the UNIX based systems
- Create processes and implement IPC
- Compare the performance of various CPU Scheduling Algorithm
- Implement deadlock avoidance, and Detection Algorithms
- analyze the performance of the various page replacement algorithm

Ex. No. 1. Basic UNIX Commands & First Program in Unix Environment

AIM :
To study and execute the commands in unix.

I. DIRECTORY RELATED COMMANDS:

1. **Present Working Directory Command :**
   To print the complete path of the current working directory.
   Syntax : $pwd

2. **MKDIR Command :**
   To create or make a new directory in a current directory.
   Syntax : $mkdir <directory name>

3. **CD Command :**
   To change or moving back or move to the mentioned directory.
   Syntax :
   ```
   $cd <directory name>.
   $cd ..
   $cd xxx/xxy
   ```

4. **RMDIR Command :**
   To remove a directory in the current directory but directory must be empty.
   Syntax : $rmdir <directory name>

II. FILE RELATED COMMANDS :

1. **CREATE A FILE :**
   To create a new file in the current directory, use CAT command.
   Syntax : $cat > filename.
   The > symbol is used in cat command.

2. **DISPLAY A FILE :**
   To display the content of file use CAT command without „>" operator.
   Syntax : $cat filename.
   Options: –s = to neglect the warning /error message.

3. **COPYING CONTENTS :**
   To copy the content of one file with another. If file doesnot exist, a new file is created and if the file exists with some data then it is overwritten.
   Syntax :
   $ cat <filename source> > <destination filename>

Options : - -n content of file with numbers included with blank lines.
Syntax : $cat –n <filename>

4. **SORTING A FILE :**
To sort the contents in alphabetical order or in reverse order.
Syntax : $sort <filename >
Option : $ sort –r <filename>

5. **COPYING CONTENTS FROM ONE FILE TO ANOTHER :**
To copy the contents from source to destination file, so that both contents are same.
Syntax : $cp <source filename> <destination filename>
$cp <source filename path > <destination filename path>

6. **MOVE Command :**
To completely move the contents from source file to destination file and to rename also.
Syntax : $ mv <source filename> <destination filename>
Ex: mv prg.c prg.bak
mv prg.bak ~/gcek

7. **REMOVE Command :**
To permanently remove the file use this command .
Syntax : $rm <filename>

8. **WORD Command :**
To list the content count of no of lines, words, characters.
Syntax : $wc<filename>
Options :
-c – to display no of characters.
-l – to display only the lines.
-w – to display the no of words.

III. OTHER COMMANDS :

1. **Date Command :**
This command is used to display the current data and time.
Syntax :
$date
$date +%ch
Options : -
a = Abbrevated weekday.
A = Full weekday.
b = Abbrevated month.
B = Full month.
c = Current day and time.
C = Display the century as a decimal number.
d = Day of the month.

D = Day in ‘mm/dd/yy’ format
h = Abbrevated month day.
H = Display the hour.
L = Day of the year.
m = Month of the year.
M = Minute.
P = Display AM or PM
S = Seconds
T = HH:MM:SS format
u = Week of the year.
y = Display the year in 2 digit.
Y = Display the full year.
Z = Time zone .

To change the format :
Syntax : $date ‘+%H-%M-%S’

2. **Calender Command :**
This command is used to display the calendar of the year or the particular month of calendar year.
Syntax : $cal <year>
$cal <month> <year>
Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

3. **who Command :**
It is used to display who are the users connected to our computer currently.
Syntax : $who – option
Options : -
H–Display the output with headers.
b–Display the last booting date or time or when the system was lastely rebooted.

4. **man Command :**
It help us to know about the particular command and its options & working. It is like help command in windows .
Syntax : $man <command name>
Ex: man ls
man printf

5. **LIST Command :**
It is used to list all the contents in the current working directory.
Syntax : $ ls – options <arguments>
If the command does not contain any argument means it shows the list of files in the current directory.

Options :

a– used to list all the files including the hidden files.
c– list all the files columnwise.
d- list all the directories.
m- list the files separated by commas.
p- list files include '/' to all the directories.
r- list the files in reverse alphabetical order.
f- list the files based on the list modification date.
x-list in column wise sorted order.

6. **CLEAR Command :**
It is used to clear the screen.
Syntax : $clear

7. **EXIT Command :**
It is used to exit the terminal.
Syntax : $exit

8. **Echo Command :**
This command is used to print the arguments on the screen .
Syntax : $echo <text>

Multi line echo command :
To have the output in the **same line**, the following command can be used.
Syntax : $echo <text\>text
To have the output in **different line**, the following commands can be used.
Syntax : $echo "text
>line2
>line3"

First C Program in Unix Environment

Aim:
To write a program in Unix Environment.

Algorithm:
Step 1: Open an editor to write the program, i.e. vi or gedi, ex: gedit prg.c
Step 2: Write the code and save it using the .c extension.
Step 3: Compile the program and generate the object file, ex: gcc –o obj prg.c
Step 4: Run the program using the object file, ex: ./obj

Ex. No. 2. UNIX Shell Programming

A) MAXIMUM OF THREE NUMBERS:

Aim:
To write a shell program to find greatest of three numbers.

Algorithm:

```
Step1: Declare the three variables.
Step2: Check if A is greater than B and C.
Step3: If so, then print A is greater.
Step4: Else check if B is greater than C.
Step5: If so print B is greater.
Step6: Else print C is greater.
```

Program:

```
echo "enter A"
read a
echo "enter B"
read b
echo "enter C"
read c
if [ $a -gt $b -a $a -gt $c ]
then
echo "A is greater"
elif [ $b -gt $a -a $b -gt $c ]
then
echo "B is greater"
else
echo "C is greater"
fi
```

RUN:

Sample I/P:

```
Enter A:50
Enter B:35
Enter C:55
```

Sample O/P:

```
C is greater
```

Result: Thus the shell program to find the maximum of three numbers is executed and the output is verified successfully.

B) FIBONACCI SERIES:

Aim:
To write a shell program to generate Fibonacci series.

Algorithm :

```
Step 1 : Initialise a to 0 and b to 1.
Step 2 : Print the values of 'a' and 'b'.
Step 3 : Add the values of 'a' and 'b'. Store the added value in variable 'c'.
Step 4 : Print the value of 'c'.
Step 5 : Initialise 'a' to 'b' and 'b' to 'c'.
Step 6 : Repeat the steps 3,4,5 till the value of 'a' is less than given number.
```

Program :

```
echo “enter the number”
read n
a=0
b=1
i=0
while [ $i – le $n ]
do
c=`expr $a + $b`
echo $c
a=$b
b=$c
i=`expr $i + 1’
done
```

Sample I/P :
Enter the no: 5

Sample O/P:

```
011235
```

Result :
Thus the shell program to find the fibonacci series is executed and output is verified successfully.

Ex. No. 3. Implement CPU scheduling algorithms

A) FIRST COME FIRST SERVED (FCFS)

Aim:
To write a c program to implement the FCFS scheduling.

Algorithm:
```
Step 1: Start
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Set the waiting of the first process as 0 and its burst time as its turn around time
Step 5: for each process in the Ready Q calculate

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

Step 6: Calculate

        Average waiting time = Total waiting Time / Number of process
        Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop
```

Program:
```
#include<stdio.h>
#include<conio.h>
int main()
{
        int bt[20], wt[20], tat[20], i, n;
        float wtavg, tatavg;
        printf("\nEnter the number of processes -- ");
                scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for Process %d -- ", i);
                scanf("%d", &bt[i]);
        }
        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("\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, bt[i], wt[i], tat[i]);
          printf("\nAverage Waiting Time -- %f", wtavg/n);
           printf("\nAverage Turnaround Time -- %f", tatavg/n);
           getch();
          return 0;

 }
```

RUN:

```
Enter the number of processes -- 4

Enter Burst Time for Process 0 -- 12

Enter Burst Time for Process 1 -- 2

Enter Burst Time for Process 2 -- 3

Enter Burst Time for Process 3 -- 5
       PROCESS      BURST TIME      WAITING TIME    TURNAROUND TIME

       P0                 12                0          12
       P1                  2               12          14
       P2                  3               14          17
       P3                  5               17          22
Average Waiting Time -- 10.750000
Average Turnaround Time -- 16.250000
```

B) SHORTEST JOB FIRST (SJF)

Aim:
Write a C program to implement the SJF Scheduling Algorithm.

Algorithm:

```
Step 1: Start
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst
time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to
lowest to highest burst time.
Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst
time.
Step 6: For each process in the ready queue, calculate

        Waiting time for process(n)= waiting time of process (n-1) + Burst time
of process (n-1)
        Turn around time for Process(n)= Turn around time of Process (n-1)+ Burst time
for process (n)
Step 6: Calculate
        Average waiting time = Total waiting Time / Number of process
        Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop
```

Program:

```
#include<stdio.h>
#include<conio.h>
main()
{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
printf("\nEnter 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;

temp=p[i];
p[i]=p[k];
p[k]=temp;
}
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);
getch();
return 0;
}
```

RUN:

```
Enter the number of processes -- 4
Enter Burst Time for Process 0 -- 12
Enter Burst Time for Process 1 -- 2
Enter Burst Time for Process 2 -- 3
Enter Burst Time for Process 3 -- 5

    PROCESS      BURST TIME     WAITING TIME    TURNAROUND TIME

     P1               2              0                2
     P2               3              2                5
     P3               5              5               10
     P0              12             10               22
     Average Waiting Time -- 4.250000
     Average Turnaround Time -- 9.750000
```

C) **ROUND ROBIN (RR)**

Aim:
Write a C program to implement the Round Robin Scheduling Algorithm.

Algorithm:

```
 Step 1: Start
 Step 2: Accept the number of processes in the ready Queue and time quantum
 Step 3: For each process in the ready Q, assign the process id and accept the CPU burst
 time
 Step 4: Calculate the no. of time slices for each process where  No. of time slice for
process(n) = burst time process(n)/time slice
 Step 5: If the burst time is less than the time slice then the no. of time slices =1.
 Step 6: Consider the ready queue is a circular Q, calculate

Waiting time for process(n) = waiting time of process(n-1)+ burst time of  process(n-1 ) +
the time difference in getting the CPU from process(n-1)

Turn around time for process (n) = waiting time of the process (n) + burst time of
process(n)+ the time difference in getting CPU from process(n).
Step 7: Calculate
Average waiting time = Total waiting Time / Number of process
Average Turnaround time = Total Turnaround Time / Number of process
Step 8: Stop
```

Program:
```
#include<stdio.h>
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 time quantum -- ");
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("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
}
```

RUN:

```
Enter the no of processes --4
Enter Burst Time for process 1 -- 12
Enter Burst Time for process 2 -- 2
Enter Burst Time for process 3 -- 3
Enter Burst Time for process 4 -- 5
Enter the time quantum -- 2
The Average Turnaround time is -- 13.250000
The Average Waiting time is -- 7.750000
PROCESS  BURST TIME     WAITING TIME   TURNAROUND TIME
1             12              10          22
2             2               2            4
3             3               8           11
4             5              11           16
```

D) PRIORITY SCHEDULING

Aim:
Write a C program to implement the Priority Scheduling Algorithm.

Algorithm:

```
Step 1: Start
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
bursttime
Step 4: Sort the ready queue according to the priority number
Step 5: Set the waiting of the first process as '0' and its burst time as its turn around time
Step 6: For each process in the Ready Q
Calculate
Waiting time for process(n)= waitingtime of process (n-1) + Burst time of process(n-1)
Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
Calculate
Average waiting time = Total waiting Time / Number of process
Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop
```

Program:

```
#include<stdio.h>
Int main()
{
        int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
        float wtavg, tatavg;
        printf("Enter the number of processes : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                p[i] = i;
                printf("Enter the Burst Time & Priority of Process %d :",i);
                scanf("%d %d",&bt[i], &pri[i]);
        }
        for(i=0;i<n;i++)
                for(k=i+1;k<n;k++)
                        if(pri[i] > pri[k])
                        {
                                temp=p[i];
                                p[i]=p[k];
                                p[k]=temp;

                                temp=bt[i];
                                bt[i]=bt[k];
                                bt[k]=temp;
```

```
                    temp=pri[i];
                    pri[i]=pri[k];
                    pri[k]=temp;

           }
wtavg = wt[0] = 0;
tatavg = tat[0] = 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\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f",tatavg/n);
getch();
return 0;
}
```

RUN:

```
Enter the number of processes : 4
Enter the Burst Time & Priority of Process 0 :12
2
Enter the Burst Time & Priority of Process 1 :2
1
Enter the Burst Time & Priority of Process 2 :3
4
Enter the Burst Time & Priority of Process 3 :5
1

PROCESS      PRIORITY     BURST TIME    WAITING TIME   TURNAROUND
    1            1            2              0             2
    3            1            5              2             7
    0            2            12             7             19
    2            4            3              19            22
    Average Waiting Time is --- 7.000000
    Average Turnaround Time is --- 12.500000
```

Ex. No. 4 Programs on System calls

Aim:
To implement the fork system call in UNIX environment.

Algorithm:
Step 1: Start
Step 2: Initialize the buffer size
Step 3: Print using the formatted printf
Step 4: Write the result
Step 5: Stop

Program:

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>
#define   MAX_COUNT  200
#define   BUF_SIZE   100
void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

RUN:

```
This line is from pid 3456, value 13
This line is from pid 3456, value 14
    ................
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
    ................
This line is from pid 3456, value 21
```

Ex. No. 5 Program to simulate the concept of Dining-Philosophers problem

Aim:

Program to simulate the concept of Dining-Philosophers problem.

Algorithm:

```
Step 1: Start
Step 2: Define the number of philosophers
Step 3: Declare one thread per philosopher
Step 4: Declare one semaphore (represent chopsticks) per philosopher
Step 5: When a philosopher is hungry
            See if chopsticks on both sides are free
            Acquire both chopsticks
            Eat
            Restore the chopstick
            If chopsticks aren't free
Step 6: Wait till they are available
Step 7: Stop
```

Program:

```
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
int i;
printf("\n\nDINING PHILOSOPHER PROBLEM");
printf("\nEnter the total no. of philosophers: ");
scanf("%d",&tph);
for(i=0;i<tph;i++)
{
philname[i] = (i+1);
status[i]=1;
}
printf("How many are hungry : ");
scanf("%d", &howhung);
if(howhung==tph)
{
printf("\nAll are hungry..\nDead lock stage will occur");
printf("\nExiting..");
}
else
{
for(i=0;i<howhung;i++)
{
printf("Enter philosopher %d position: ",(i+1));
```

```
scanf("%d", &hu[i]);
status[hu[i]]=2;
}
do
{
printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:");
scanf("%d", &cho);
switch(cho)
{
case 1: one();
        break;
case 2: two();
        break;
case 3: exit(0);
default: printf("\nInvalid option..");
}
}while(1);
}
}

one()
{
        int pos=0, x, i;
        printf("\nAllow one philosopher to eat at any time\n");
        for(i=0;i<howhung; i++, pos++)
        {
                printf("\nP %d is granted to eat", philname[hu[pos]]);
                for(x=pos;x<howhung;x++)
                    printf("\nP %d is waiting", philname[hu[x]]);
        }
}
two()
{
        int i, j, s=0, t, r, x;
        printf("\n Allow two philosophers to eat at same time\n");
        for(i=0;i<howhung;i++)
        {
                for(j=i+1;j<howhung;j++)
                {
                        if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
                        {
                                printf("\n\ncombination %d \n", (s+1));
                                t=hu[i];
                                r=hu[j];
                                s++;
            printf("\nP %d and P %d are granted to eat", philname[hu[i]], philname[hu[j]]);

                                for(x=0;x<howhung;x++)
                                {
                                        if((hu[x]!=t)&&(hu[x]!=r))
                                        printf("\nP %d is waiting", philname[hu[x]]);
                                }
                        }
                }
        }
}
```

RUN:

```
1.One can eat at a time 2.Two can eat at a time 3.Exit
Enter your choice: 1

Allow one philosopher to eat at any time
P 3 is granted to eat
P 3 is waiting
P 5 is waiting
P 0 is waiting
P 5 is granted to eat
P 5 is waiting
P 0 is waiting
P 0 is granted to eat
P 0 is waiting
1. One can eat at a time 2.Two can eat at a time 3.Exit Enter
your choice: 2

 Allow two philosophers to eat at same time
combination 1
P 3 and P 5 are granted to eat
P 0 is waiting

combination 2
P 3 and P 0 are granted to eat
P 5 is waiting

combination 3
P 5 and P 0 are granted to eat
P 3 is waiting

1.One can eat at a time 2.Two can eat at a time 3.Exit Enter
your choice: 3
```

Ex. No. 6 Bankers Algorithm for Dead Lock Avoidance

Aim:
To implement deadlock avoidance & Prevention by using Banker‟s Algorithm.

Algorithm:

```
Step 1: Start
Step 2: Get the values of resources and processes.
Step 3: Get the avail value.
Step 4: After allocation find the need value.
Step 5: Check whether its possible to allocate.
Step 6: If it is possible then the system is in safe state.
Step 7: Else system is not in safety state.
Step 8: If the new request comes then check that the system is in safety or not if we allow request.
Step 9: Stop
```

Program:

```
#include<stdio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};
int main()
{
        struct file f[10];
        int fl;
        int i, j, k, p, b, n, r, g, cnt=0, id, newr;
        int avail[10],seq[10];
        printf("Enter number of processes -- ");
        scanf("%d",&n);
        printf("Enter number of resources -- ");
        scanf("%d",&r);
        for(i=0;i<n;i++)
        {
                printf("Enter details for P%d",i);
                printf("\nEnter allocation\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].all[j]);
                        printf("Enter Max\t\t -- \t");
                for(j=0;j<r;j++)
```

```
                    scanf("%d",&f[i].max[j]);
          f[i].flag=0;
}
printf("\nEnter Available Resources\t -- \t");
for(i=0;i<r;i++)
           scanf("%d",&avail[i]);

printf("\nEnter New Request Details -- ");
printf("\nEnter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t -- \t");
for(i=0;i<r;i++)
{
          scanf("%d",&newr);
          f[id].all[i] += newr;
          avail[i]=avail[i] - newr;


  }

for(i=0;i<n;i++)
{
          for(j=0;j<r;j++)
          {
                    f[i].need[j]=f[i].max[j]-f[i].all[j];
                    if(f[i].need[j]<0)
                              f[i].need[j]=0;
          }
}
cnt=0;
fl=0;
while(cnt!=n)
{
          g=0;
          for(j=0;j<n;j++)
          {
                    if(f[j].flag==0)
                    {
                              b=0;
                              for(p=0;p<r;p++)
                              {
                                        if(avail[p]>=f[j].need[p])
                                                    b=b+1;
                                           else
                                                   b=b-1;
```

```
                        }

                        if(b==r)
                        {
                                printf("\nP%d is visited",j);
                                seq[fl++]=j;
                                f[j].flag=1;
                                for(k=0;k<r;k++)
                                        avail[k]=avail[k]+f[j].all[k];
                                cnt=cnt+1;
                                printf("(");
                                for(k=0;k<r;k++)
                                        printf("%3d",avail[k]);
                                printf(")");
                                g=1;
                        }
                }
        }
        if(g==0)
        {
                printf("\n REQUEST NOT GRANTED -- DEADLOCK
                OCCURRED"); printf("\n SYSTEM IS IN UNSAFE
                STATE"); goto y;

        }
    }
    printf("\nSYSTEM IS IN SAFE STATE");
    printf("\nThe Safe Sequence is -- (");
    for(i=0;i<fl;i++)
            printf("P%d ",seq[i]);
    printf(")");
  y:printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
    for(i=0;i<n;i++)
    {
            printf("P%d\t",i);
            for(j=0;j<r;j++)
                    printf("%6d",f[i].all[j]);
            for(j=0;j<r;j++)
                    printf("%6d",f[i].max[j]);
            for(j=0;j<r;j++)
                    printf("%6d",f[i].need[j]);
            printf("\n");
    }
    getch();
    return 0;       }
```

RUN:

```
INPUT
Enter number of processes          –      5
Enter number of resources          --     3
Enter details for P0
Enter
allocation                   --    0       1      0
Enter Max                    --           7      5      3

Enter details for P1
Enter
allocation                   --    2      0      0
Enter Max                    --    3      2      2

Enter details for P2
Enter
allocation                   --    3      0      2
Enter Max                    --    9      0      2

Enter details for P3
Enter
allocation                   --    2      1      1
Enter Max                    --    2      2      2

Enter details for P4
Enter
allocation                   --    0      0      2
Enter Max                    --    4      3      3

Enter Available Resources --    3  3      2
Enter New Request Details --
Enter pid     --    1
Enter Request for Resources     --  1        0     2

OUTPUT
P1 is visited( 5  3   2)
P3 is visited( 7  4   3)
P4 is visited( 7  4   5)
P0 is visited( 7  5   5)
P2 is visited( 10   5  7)
SYSTEM IS IN SAFE
STATE
The Safe Sequence is -- (P1 P3 P4 P0 P2 )

Process         Allocation              Max                 Need
P0              0    1    0             7    5    3         7   4 3
P1              3    0    2             3    2    2         0   2 0
P2              3    0    2             9    0    2         6   0 0
P3              2    1    1             2    2    2         0   1 1
P4              0    0    2             4    3    3         4   3 1
```

Ex. No. 7 Implement page replacement algorithms:

A) FIRST IN FIRST OUT (FIFO)

Aim:
Write a C program to implement the First In First Out Algorithm.

Algorithm:

```
Step 1: Start
Step 2: Create a queue to hold all pages in memory
Step 3: When the page is required replace the page at the head of the queue
Step 4: Now the new page is inserted at the tail of the queue
Step 5: Stop
```

Program:

```
#include<stdio.h>
#include<conio.h>
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);
        getch();
        }
```

RUN:

```
Enter the length of reference string -- 10

 Enter the reference string -- 2 3 4 5 2 3 6 3 1 8

 Enter no. of frames -- 3

 The Page Replacement Process is --
        2       -1      -1      PF No. 1
        2       3       -1      PF No. 2
        2       3       4       PF No. 3
        5       3       4       PF No. 4
        5       2       4       PF No. 5
        5       2       3       PF No. 6
        6       2       3       PF No. 7
        6       2       3
        6       1       3       PF No. 8
        6       1       8       PF No. 9

 The number of Page Faults using FIFO are 9
--------------------------------
Process exited after 105.6 seconds with return value 1
Press any key to continue . . .
```

B) LEAST RECENTLY USED (LRU)

Aim:
Write a C program to implement the Least Recently Used Algorithm.
Algorithm:
Step 1: Start
Step 2: Create a queue to hold all pages in memory
Step 3: When the page is required replace the page at the head of the queue
Step 4: Now the new page is inserted at the tail of the queue
Step 5: Create a stack
Step 6: When the page fault occurs replace page present at the bottom of the stack
Step 7: Stop

Program:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
        printf("Enter the length 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 number of frames -- ");
        scanf("%d",&f);
        for(i=0;i<f;i++)
        {
                count[i]=0;
                m[i]=-1;
        }
        printf("\nThe Page Replacement process is -- \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("PF No. -- %d" , pf);
                    printf("\n");
            }
            printf("\nThe number of page faults using LRU are %d",pf);
            getch();
    }
```

RUN:

```
Enter the length of reference string -- 10
Enter the reference string -- 2 3 4 5 2 7 8 1 2 4
Enter the number of frames -- 3
The Page Replacement process is --
2     -1    -1     PF No. -- 1
2     3     -1     PF No. -- 2
2     3     4      PF No. -- 3
5     3     4      PF No. -- 4
5     2     4      PF No. -- 5
5     2     7      PF No. -- 6
8     2     7      PF No. -- 7
8     1     7      PF No. -- 8
8     1     2      PF No. -- 9
4     1     2      PF No. -- 10

The number of page faults using LRU are 10
```

C) OPTIMAL Page Replacement

Aim:
Write a C program to implement the Optimal Page Replacement Algorithm.
Algorithm:

Step 1: Start
Step 2: Create a array
Step 3: When the page fault occurs replace page that will not be used for the longest period
Step 4: Stop

Program:

```
#include<stdio.h>
#include<conio.h>
int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1;
int recent[10],optcal[50],count=0;
int optvictim();
void main()
{
printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHN");
printf("\n................................");
printf("\nEnter the no.of frames");
scanf("%d",&nof);
printf("Enter the no.of reference string");
scanf("%d",&nor);
printf("Enter the reference string");
for(i=0; i<nor; i++)
   scanf("%d",&ref[i]);
printf("\n OPTIMAL PAGE REPLACEMENT ALGORITHM");
printf("\n................................");
printf("\nThe given string");
printf("\n....................\n");
for(i=0;i<nor;i++)
     printf("%4d",ref[i]);
for(i=0;i<nof;i++)
{
frm[i]=-1;
optcal[i]=0;
}
for(i=0;i<10;i++)
     recent[i]=0;
printf("\n");
```

```
for(i=0;i<nor;i++)
{
   flag=0;
    printf("\n\tref no %d ->\t",ref[i]);
    for(j=0;j<nof;j++)
     {
       if(frm[j]==ref[i])
         {
         flag=1;
         break;
          }
       }
   if(flag==0)
     {
      count++;
      if(count<=nof)
      victim++;
      else
    victim=optvictim(i);
       pf++;
      frm[victim]=ref[i];
      for(j=0;j<nof;j++)
        printf("%4d",frm[j]);
       }
 }
printf("\n Number of page faults: %d",pf);
getch();
}

int optvictim(int index)
{
int i,j,temp,notfound;
for(i=0;i<nof;i++)
   {
     notfound=1;
    for(j=index;j<nor;j++)
   if(frm[i]==ref[j])
    {
    notfound=0;
   optcal[i]=j;
    break;
     }
if(notfound==1)
return i;
   }
temp=optcal[0];

for(i=1;i<nof;i++)
  if(temp<optcal[i])
     temp=optcal[i];
for(i=0;i<nof;i++)
  if(frm[temp]==frm[i])
            return i;
return 0;
}
```

RUN:

```
 OPTIMAL PAGE REPLACEMENT ALGORITHN
.................................
Enter the no.of frames3
Enter the no.of reference string10
Enter the reference string2 3 4 5 2 7 8 1 2 4

 OPTIMAL PAGE REPLACEMENT ALGORITHM
................................
The given string
....................
   2   3   4   5   2   7   8   1   2   4

        ref no 2 ->        2  -1  -1
        ref no 3 ->        2   3  -1
        ref no 4 ->        2   3   4
        ref no 5 ->        2   5   4
        ref no 2 ->
        ref no 7 ->        2   7   4
        ref no 8 ->        2   8   4
        ref no 1 ->        2   1   4
        ref no 2 ->
        ref no 4 ->
 Number of page faults: 7
```