



Degree College

Computer Journal CERTIFICATE

SEMESTER _____ II _____ UID No. _____ 2020858

Class _____ FYBSC[CS] _____ Roll No. _____ 1146 _____ Year _____ 2020-21

This is to certify that the work entered in this journal
is the work of Mst. / Ms. _____ Shaikh Kaysan Razauddin

who has worked for the year _____ 2020-21 _____ in the Computer
Laboratory.

Teacher In-Charge

Head of Department

Date : _____

Examiner

Index

Sr no.	Title	Pg. no.
1	First Come First Serve (FCFS)	3
2	Shortest Job First (SJF)	5
3	Round Robin (R.R)	7
4	Banker's algorithm	9
5	First in first out (FIFO) page replacement algorithm	14
6	Least Recently Used (LRU) page replacement algorithm	16
7	Optimal page replacement algorithm	19
8	Producer consumer problem	22
9	Dining philosophers problem	26

Practical No. 01

Aim:

To study First Come First Serve CPU scheduling

Theory:

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve. As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

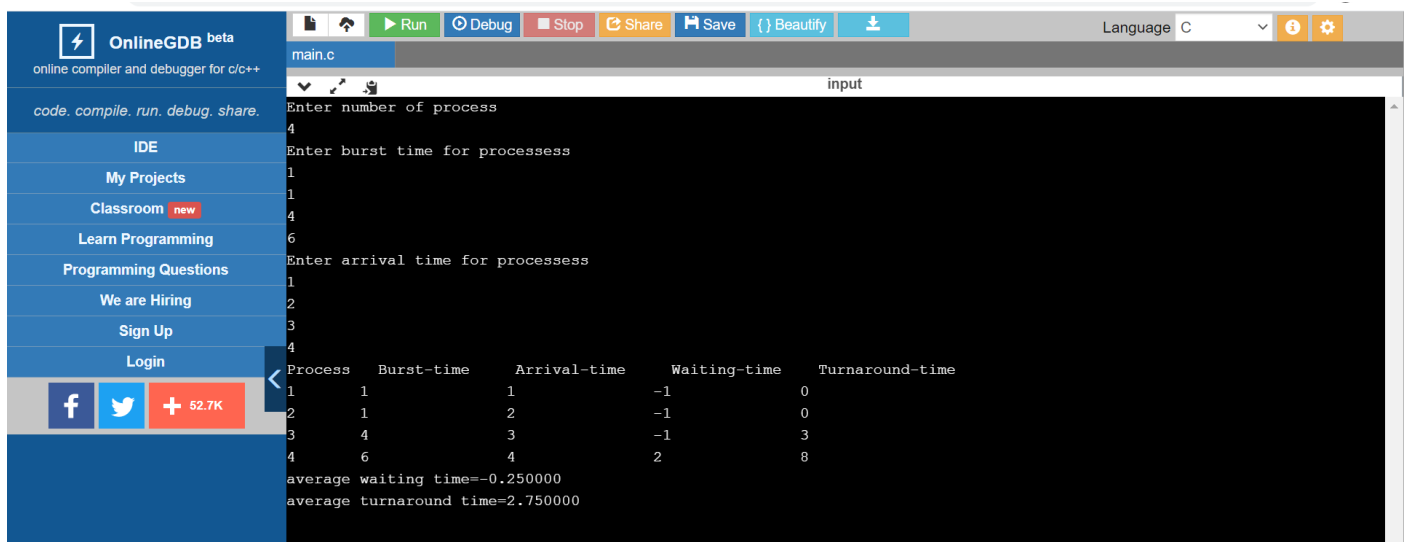
Source code:

\\WAP to implement average waiting time and turnaround time for FCFS scheduling algorithm

```
#include<stdio.h>
#include<conio.h>
#define max 30
void main()
{
    int i,j,n,bt[max],at[max],wt[max],tat[max],temp[max];
    float awt=0,atat=0;
    clrscr();
    printf("Enter number of process\n");
    scanf("%d",&n);
    printf("Enter burst time for processess\n");
    for(i=0;i<n;i++)
        scanf("%d",&bt[i]);
    printf("Enter arrival time for processess\n");
    for(i=0;i<n;i++)
        scanf("%d",&at[i]);
    temp[0]=0;
    printf("Process\t Burst-time\t Arrival-time\t Waiting-time\t Turnaround-time\n");
    for(i=0;i<n;i++)
    {
        wt[i]=0;
        tat[i]=0;
        temp[i+1]=temp[i]+bt[i];
        wt[i]=temp[i]-at[i];
        tat[i]=wt[i]+bt[i];
```

```
        awt=awt+wt[i];
        atat=atat+tat[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n",i+1,bt[i],at[i],wt[i],tat[i]);
    }
    awt=awt/n;
    atat=atat/n;
    printf("average waiting time=%f\n",awt);
    printf("average turnaround time=%f\n",atat);
    getch();
}
```

Output:



The screenshot displays the OnlineGDB beta IDE interface. The main window shows the execution of a C program. The input provided is as follows:

```
Enter number of process
4
Enter burst time for processes
1
1
4
6
Enter arrival time for processes
1
2
3
4
```

The output of the program is a table showing the scheduling details for four processes:

Process	Burst-time	Arrival-time	Waiting-time	Turnaround-time
1	1	1	-1	0
2	1	2	-1	0
3	4	3	-1	3
4	6	4	2	8

Below the table, the program calculates the average values:

```
average waiting time=-0.250000
average turnaround time=2.750000
```

Conclusion:

Hence completed the experiment for First Come First Serve (FCFS) successfully.

Practical No. 02

Aim:

To study Shortest Job First (SJF) CPU scheduling

Theory:

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

Source code:

```
#include<stdio.h>
#include<conio.h>
#define max 30
void main()
{
    int i,j,n,t,bt[max],p[max],wt[max],tat[max];
    float awt=0,atat=0;
    clrscr();
    printf("Enter number of process\n");
    scanf("%d",&n);
    printf("Enter the process number\n");
    for(i=0;i<n;i++)
        scanf("%d", &p[i]);
    printf("Enter burst time for processess\n");
    for(i=0;i<n;i++)
        scanf("%d",&bt[i]);
    for(i=0;i<n;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(bt[j]>bt[j+1])
            {
                t=bt[j];
                bt[j]=bt[j+1];
                bt[j+1]=t;

                t=p[j];
                p[j]=p[j+1];
                p[j+1]=t;
            }
        }
    }
}
```

```

printf("Process\t Burst-time\t Waiting-time\t Turnaround-time\n");
for(i=0;i<n;i++)
{
    wt[i]=0;
    tat[i]=0;
    for(j=0;j<i;j++)
    {
        wt[i]=wt[i]+bt[j];
    }
    tat[i]=wt[i]+bt[i];
    awt=awt+wt[i];
    atat=atat+tat[i];
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",p[i],bt[i],wt[i],tat[i]);
}
awt=awt/n;
atat=atat/n;
printf("average waiting time=%f\n",awt);
printf("average turnaround time=%f\n",atat);
getch();
}

```

Output:

```

main.c
1 #include<stdio.h>
...
main.c:45:34: warning: format '%d' expects a matching 'int' argument [-Wformat=]
Enter number of process
4
Enter the process number
1
1
4
6
Enter burst time for processess
1
2
2
3
Process  Burst-time  Waiting-time  Turnaround-time
1        1           0              1      1701669236
1        2           1              3      1701669236
4        2           3              5      1701669236
6        3           5              8      1701669236
average waiting time=2.250000
average turnaround time=4.250000

```

Conclusion:

Hence completed the experiment for Shortest Job First (SJF) successfully.

Practical No. 03

Aim:

To study Round Robin (R.R) CPU scheduling

Theory:

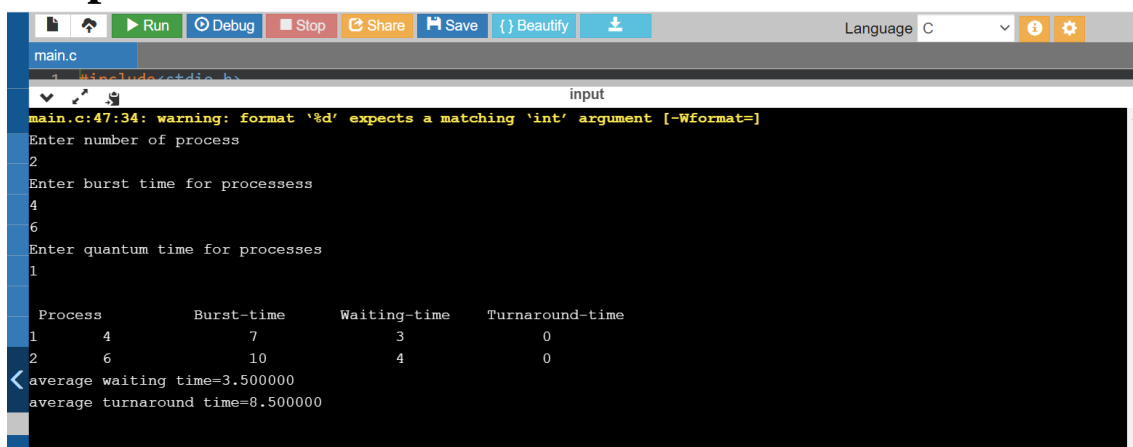
A round-robin is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called time quantum or time slice. Some of the round-robin processes are pre-empted if it executed in a given time slot, while the rest of the processes go back to the ready queue and wait to run in a circular order with the scheduled time slot until they complete their task. It removes the starvation for each process to achieve CPU scheduling by proper partitioning of the CPU.

Source code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,qt,count=0,temp,sq=0,bt[10],wt[10],tat[10],rem_bt[10];
    float awt=0,atat=0;
    clrscr();
    printf("Enter number of process\n");
    scanf("%d",&n);
    printf("Enter burst time for processess\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&bt[i]);
        rem_bt[i]=bt[i];
    }
    printf("Enter quantum time for processes\n");
    scanf("%d",&qt);
    while(1)
    {
        for(i=0;i<n;i++)
        {
            temp=qt;
            if(rem_bt[i]==0)
            {
```

```
        count++;
        continue;
    }
    if(rem_bt[i]>qt)
        rem_bt[i]=rem_bt[i]-qt;
    else
        if(rem_bt[i]>0)
        {
            temp=rem_bt[i];
            rem_bt[i]=0;
        }
        sq=sq+temp;
        tat[i]=sq;
    }
    if(n==count)
        break;
}
printf("\n Process\t Burst-time\t Waiting-time\t Turnaround-time\n");
for(i=0;i<n;i++)
{
    wt[i]=tat[i]-bt[i];
    awt=awt+wt[i];
    atat=atat+tat[i];
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",i+1,bt[i],tat[i],wt[i]);
}
awt=awt/n;
atat=atat/n;
printf("average waiting time=%f\n",awt);
printf("average turnaround time=%f\n",atat);
getch();
}
```

Output:



```
main.c
#include<stdio.h>
int main()
{
    int n, bt[10], tat[10], wt[10], rem_bt[10], sq=0, qt, i, j, count=0, awt=0, atat=0;
    printf("Enter number of process\n");
    scanf("%d", &n);
    printf("Enter burst time for processess\n");
    for(i=0; i<n; i++)
    {
        scanf("%d", &bt[i]);
    }
    printf("Enter quantum time for processes\n");
    scanf("%d", &qt);
    printf("\n Process\t Burst-time\t Waiting-time\t Turnaround-time\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i+1, bt[i], tat[i], wt[i], rem_bt[i], sq);
        sq=0;
    }
    printf("average waiting time=%f\n", awt/n);
    printf("average turnaround time=%f\n", atat/n);
    getch();
}
```

main.c:47:34: warning: format '%d' expects a matching 'int' argument [-Wformat=]

Enter number of process
2
Enter burst time for processess
4
6
Enter quantum time for processes
1

Process	Burst-time	Waiting-time	Turnaround-time
1	4	7	3
2	6	10	4

average waiting time=3.500000
average turnaround time=8.500000

Conclusion:

Hence completed the experiment for Round robin (R.R) successfully.

Practical No .04

Aim:

To study Banker's algorithm.

Theory:

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources.

Source code:

```
// Banker's Algorithm
#include <stdio.h>
#include <conio.h>
void main()
{
    int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];
    int p, r, i, j, process, count;
    count = 0;
    printf("Enter the no of processes: ");
    scanf("%d", &p);
    for(i = 0; i < p; i++)
        completed[i] = 0;
    printf("\n\nEnter the no of resources: ");
    scanf("%d", &r);
    printf("\n\nEnter the Max Matrix for each process: ");
    for(i = 0; i < p; i++)
```

```
{  
    printf("\nFor process %d: ", i + 1);  
    for(j = 0; j < r; j++)  
        scanf("%d", &Max[i][j]);  
}  
printf("\n\nEnter the allocation for each process : ");  
for(i = 0; i < p; i++)  
{  
    printf("\nFor process %d: ", i + 1);  
    for(j = 0; j < r; j++)  
        scanf("%d", &alloc[i][j]);  
}  
printf("\n\nEnter the Available Resources: ");  
for(i = 0; i < r; i++)  
    scanf("%d", &avail[i]);  
for(i = 0; i < p; i++)  
    for(j = 0; j < r; j++)  
        need[i][j] = Max[i][j] - alloc[i][j];  
do  
{  
    printf("\n Max matrix:\tAllocation matrix:\n");  
    for(i = 0; i < p; i++)  
    {  
        for( j = 0; j < r; j++)  
            printf("%d ", Max[i][j]);  
        printf("\t\t");  
        for( j = 0; j < r; j++)  
            printf("%d ", alloc[i][j]);  
        printf("\n");  
    }  
}
```

```
process = -1;

for(i = 0; i < p; i++)
{
    if(completed[i] == 0)//if not completed
    {
        process = i ;
        for(j = 0; j < r; j++)
        {
            if(avail[j] < need[i][j])
            {
                process = -1;
                break;
            }
        }
        if(process != -1)
            break;
    }
    if(process != -1)
    {
        printf("\nProcess %d runs to completion!", process + 1);
        safeSequence[count] = process + 1;
        count++;
        for(j = 0; j < r; j++)
        {
            avail[j] += alloc[process][j];
            alloc[process][j] = 0;
            Max[process][j] = 0;
            completed[process] = 1;
        }
    }
}
```

```
    }  
}  
  
while(count != p && process != -1);  
  
if(count == p)  
  
{  
    printf("\nThe system is in a safe state!!\n");  
    printf("Safe Sequence : < ");  
    for( i = 0; i < p; i++)  
        printf("%d ", safeSequence[i]);  
    printf(">\n");  
}  
  
else  
    printf("\nThe system is in an unsafe state!!");  
getch();  
  
}
```

Output:

```
input
Enter the no of processes: 2
Enter the no of resources: 2
Enter the Max Matrix for each process:
For process 1: 1
1
For process 2: 4
6
Enter the allocation for each process :
For process 1: 2
3
For process 2: 4
5
Enter the Available Resources: 4
5
Max matrix:      Allocation matrix:
1  1             2  3
4  6             4  5
Process 1 runs to completion!
Max matrix:      Allocation matrix:
0  0             0  0
4  6             4  5
Process 2 runs to completion!
The system is in a safe state!!
Safe Sequence : < 1 2 >
```

Conclusion:

Hence completed the experiment for Banker's algorithm successfully.

Practical No .05

Aim:

To study First in First out page replace

Theory:

When a page fault occurs, the OS has to remove a page from the memory so that it can fit in another page in the memory. These page replacement algorithms are used in operating systems that support virtual memory management. FIFO Page Replacement technique is one of the simplest one to implement amongst other page replacement algorithms. It is a conservative algorithm. It is a low-overhead algorithm that maintains a queue to keep a track of all the pages in a memory. When a page needs to be replaced, the page at the FRONT of the Queue will be replaced. The FIFO page replacement technique is not implemented in operating systems nowadays.

Source code:

C program to implement FIFO page replacement algorithm

```
#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
    printf("\nEnter the length of the Reference string:\n");
    scanf("%d",&n);
    printf("\nEnter the reference string:\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    printf("\nEnter the number of Frames:");
    scanf("%d",&no);
    for(i=0;i<no;i++)
        frame[i]= -1;
        j=0;
        printf("\nRef string\t page frames\t Hit/Fault\n");
        for(i=1;i<=n;i++)
        {
            printf("%d\t",a[i]);
            avail=0;
            for(k=0;k<no;k++)
                if(frame[k]==a[i])
                {
                    avail=1;
                    for(k=0;k<no;k++)
                        printf("%d\t",frame[k]);
                    printf("H");
                }
            if (avail==0)
```

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

Output:

```
enter the length of the Reference string:  
6  
  
enter the reference string:  
1  
1  
4  
6  
1  
2  
  
enter the number of Frames:3  
  
    ref string      page frames      Hit/Fault  
1          1         -1         -1         F  
1          1         -1         -1         H  
4          1          4         -1         F  
6          1          4          6         F  
1          1          4          6         H  
2          2          4          6         F  
Page Fault is 4
```

Conclusion:

Hence completed the experiment for First in first out (FIFO) successfully.

Practical No .06

Aim:

To study Least Recently Used (LRU) page replace

Theory:

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page is not found in the memory frames.

Source code:

```
#include<stdio.h>
#include<conio.h>

int findLRU(int time[], int n)
{
    int i, minimum = time[0], pos = 0;

    for(i = 1; i < n; ++i)
    {
        if(time[i] < minimum)
        {
            minimum = time[i];
            pos = i;
        }
    }

    return pos;
}

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
    i, j, pos, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);
```



```
printf("Enter number of pages: ");  
scanf("%d", &no_of_pages);
```

```
printf("Enter reference string: ");
```

```
for(i = 0; i < no_of_pages; ++i)
```

```
{  
    scanf("%d", &pages[i]);  
}
```

```
for(i = 0; i < no_of_frames; ++i)
```

```
{  
    frames[i] = -1;  
}
```

```
for(i = 0; i < no_of_pages; ++i)
```

```
{  
    flag1 = flag2 = 0;  
  
    for(j = 0; j < no_of_frames; ++j)  
    {  
        if(frames[j] == pages[i])  
        {  
            counter++;  
            time[j] = counter;  
            flag1 = flag2 = 1;  
            break;  
        }  
    }  
}
```

```
if(flag1 == 0)
```

```
{  
    for(j = 0; j < no_of_frames; ++j)  
    {  
        if(frames[j] == -1)  
        {  
            counter++;  
            faults++;  
            frames[j] = pages[i];  
            time[j] = counter;  
            flag2 = 1;  
            break;  
        }  
    }  
}
```

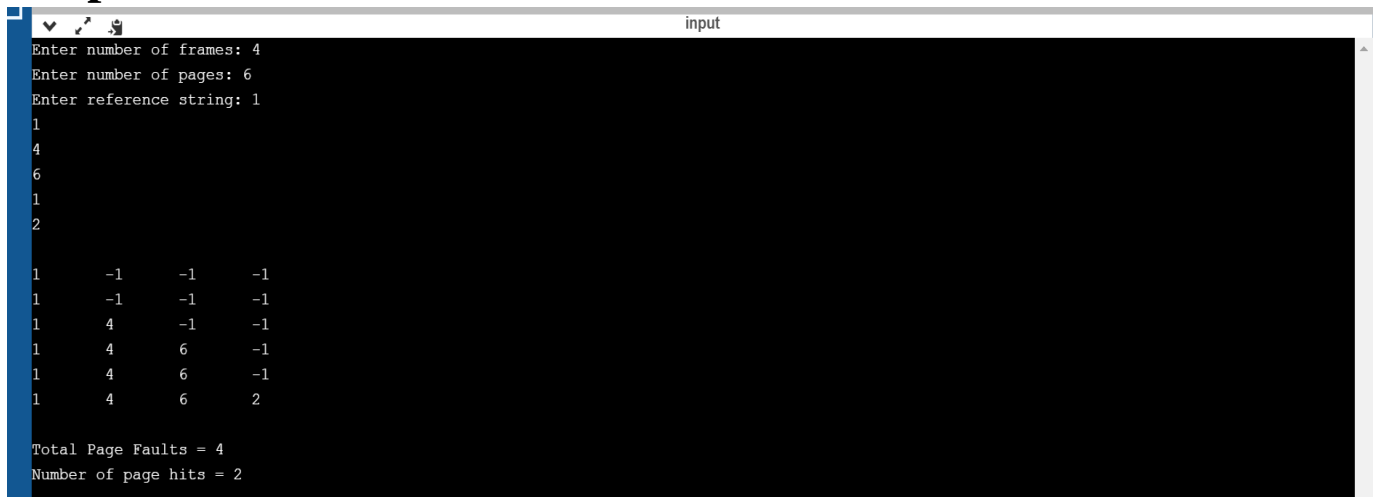
```
if(flag2 == 0)
{
    pos = findLRU(time, no_of_frames);
    counter++;
    faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
}

printf("\n");

for(j = 0; j < no_of_frames; ++j)
{
    printf("%d\t", frames[j]);
}

printf("\n\nTotal Page Faults = %d", faults);
printf("\nNumber of page hits = %d", (no_of_pages-faults));
return 0;
}
```

Output:



```
input
Enter number of frames: 4
Enter number of pages: 6
Enter reference string: 1
1
4
6
1
2

1      -1      -1      -1
1      -1      -1      -1
1       4       -1      -1
1       4       6       -1
1       4       6       -1
1       4       6       2

Total Page Faults = 4
Number of page hits = 2
```

Conclusion:

Hence completed the experiment for Least Recently Used (LRU) page replace successfully.

Practical No .07

Aim:

To study optimal page replacement.

Theory:

Optimal Page Replacement algorithm algorithms replaces the page which will not be referred for so long in future. Although it can not be practically implementable but it can be used as a benchmark. Other algorithms are compared to this in terms of optimality.

Source code:

```
#include<stdio.h>
int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos,
    max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                flag1 = flag2 = 1;
                break;
            }
        }
    }
}
```

```
if(flag1 == 0){
    for(j = 0; j < no_of_frames; ++j){
        if(frames[j] == -1){
            faults++;
            frames[j] = pages[i];
            flag2 = 1;
            break;
        }
    }
}

if(flag2 == 0){
    flag3 = 0;

    for(j = 0; j < no_of_frames; ++j){
        temp[j] = -1;

        for(k = i + 1; k < no_of_pages; ++k){
            if(frames[j] == pages[k]){
                temp[j] = k;
                break;
            }
        }
    }

    for(j = 0; j < no_of_frames; ++j){
        if(temp[j] == -1){
            pos = j;
            flag3 = 1;
            break;
        }
    }

    if(flag3 == 0){
        max = temp[0];
        pos = 0;

        for(j = 1; j < no_of_frames; ++j){
            if(temp[j] > max){
                max = temp[j];
                pos = j;
            }
        }

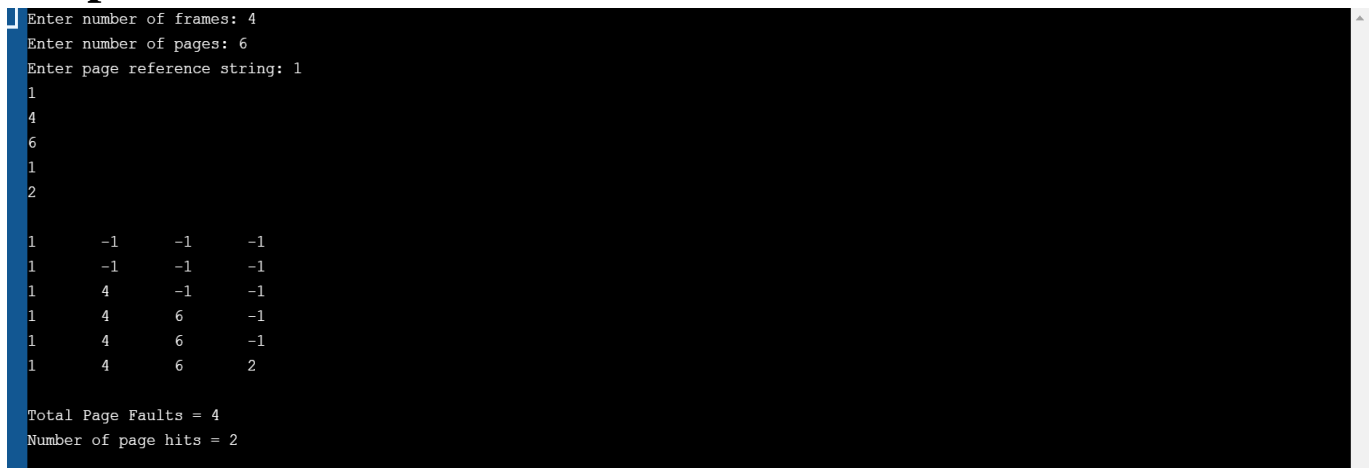
        frames[pos] = pages[i];
        faults++;
    }
}
```

```
printf("\n");

for(j = 0; j < no_of_frames; ++j){
    printf("%d\t", frames[j]);
}

printf("\n\nTotal Page Faults = %d", faults);
printf("\nNumber of page hits = %d", (no_of_pages-faults));
return 0;
}
```

Output:



```
Enter number of frames: 4
Enter number of pages: 6
Enter page reference string: 1
1
4
6
1
1
2

1      -1      -1      -1
1      -1      -1      -1
1      4       -1      -1
1      4       6       -1
1      4       6       -1
1      4       6       2

Total Page Faults = 4
Number of page hits = 2
```

Conclusion:

Hence completed the experiment for Optimal page replace successfully.

Practical No .08

Aim:

To study Producer consumer problem

Theory:

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes. In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
// Initialize a mutex to 1
int mutex = 1;
// Number of full slots as 0
int full = 0;
// Number of empty slots as size of buffer
int empty = 10, x = 0;
// Function to produce an item and add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;
    // Increase the number of full
    // slots by 1
    ++full;
    // Decrease the number of empty
    // slots by 1
```

```
--empty;

// Item produced

x++;

printf("\nProducer produces"
        "item %d",
        x);

// Increase mutex value by 1

++mutex;

}

// Function to consume an item and remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;

    // Decrease the number of full
    // slots by 1
    --full;

    // Increase the number of empty
    // slots by 1
    ++empty;

    printf("\nConsumer consumes "
            "item %d",
            x);

    x--;

    // Increase mutex value by 1
    ++mutex;

}

// Driver Code

int main()
{
    int n, i;
```

```
printf("\n1. Press 1 for Producer"
```

```
"\n2. Press 2 for Consumer"
```

```
"\n3. Press 3 for Exit");
```

```
// Using '#pragma omp parallel for'
```

```
// can give wrong value due to synchronisation issues.
```

```
// 'critical' specifies that code is executed by only one thread at a time i.e., only one thread enters the critical section at a given time
```

```
#pragma omp critical
```

```
for (i = 1; i > 0; i++) {
```

```
    printf("\nEnter your choice:");
```

```
    scanf("%d", &n);
```

```
    // Switch Cases
```

```
    switch (n) {
```

```
    case 1:
```

```
        // If mutex is 1 and empty
```

```
        // is non-zero, then it is
```

```
        // possible to produce
```

```
        if ((mutex == 1)
```

```
            && (empty != 0)) {
```

```
            producer();
```

```
        }
```

```
        // Otherwise, print buffer
```

```
        // is full
```

```
        else {
```

```
            printf("Buffer is full!");
```

```
        }
```

```
        break;
```

```
    case 2:
```

```
        // If mutex is 1 and full
```

```
        // is non-zero, then it is
```

```
        // possible to consume
```



```

        if ((mutex == 1)
            && (full != 0)) {
            consumer();

        }
        // Otherwise, print Buffer
        // is empty
        else {
            printf("Buffer is empty!");
        }
        break;
    // Exit Condition
    case 3:
        exit(0);
        break;
}
}
}

```

Output:

```
input
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:2
Buffer is empty!
Enter your choice:1

Producer produces item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:1

Producer produces item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:2
Buffer is empty!
Enter your choice:3
```

Conclusion:

Hence completed the experiment for Producer consumer problem successfully.

Practical No .09

Aim:

To study Dining philosophers problem

Theory:

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Source code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
```

```
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n",
               phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{
    sem_wait(&mutex);
    // state that hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{
    sem_wait(&mutex);
    // state that thinking
```

```
state[phnum] = THINKING;

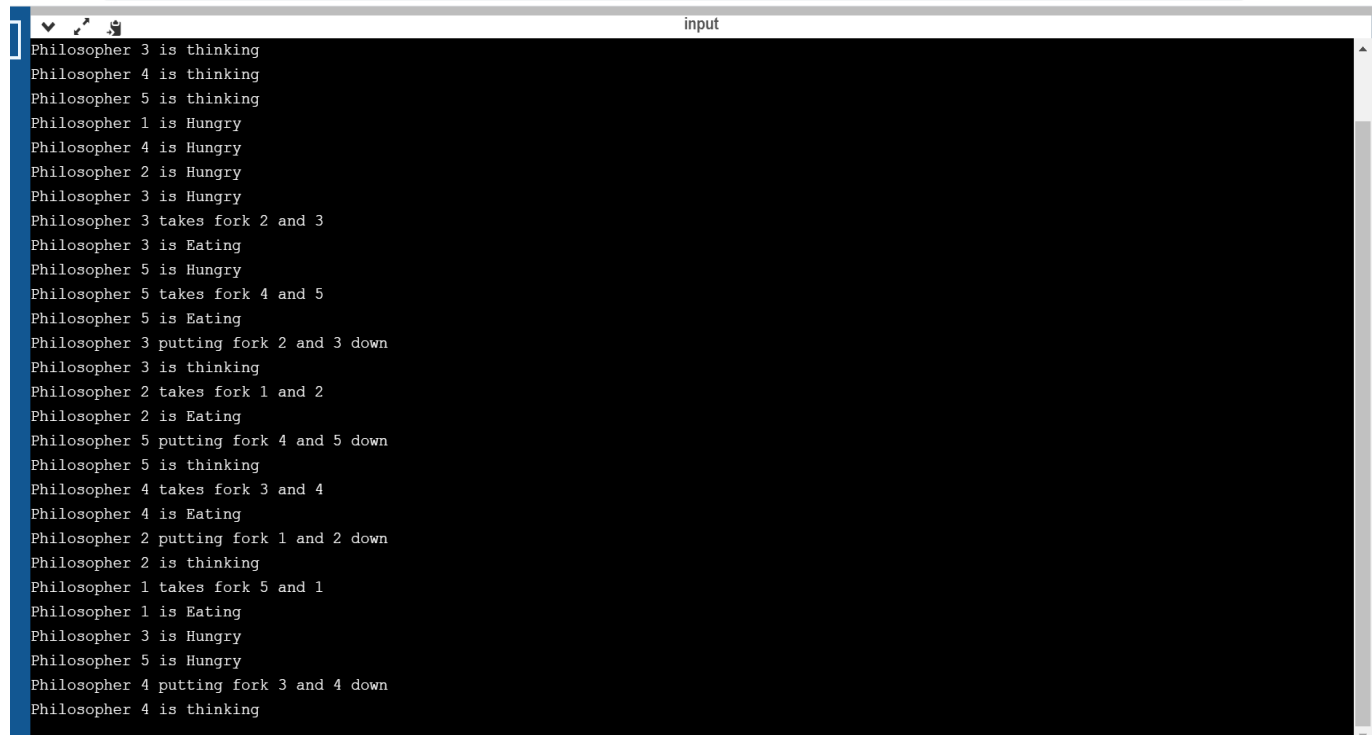
printf("Philosopher %d putting fork %d and %d down\n",
      phnum + 1, LEFT + 1, phnum + 1);
printf("Philosopher %d is thinking\n", phnum + 1);
test(LEFT);
test(RIGHT);
sem_post(&mutex);
}

void* philosopher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];
    // initialize the semaphores
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
    for (i = 0; i < N; i++) {
        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                      philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
}
```

```
}  
  
    for (i = 0; i < N; i++)  
  
        pthread_join(thread_id[i], NULL);  
  
}
```

Output:



```
Philosopher 3 is thinking  
Philosopher 4 is thinking  
Philosopher 5 is thinking  
Philosopher 1 is Hungry  
Philosopher 4 is Hungry  
Philosopher 2 is Hungry  
Philosopher 3 is Hungry  
Philosopher 3 takes fork 2 and 3  
Philosopher 3 is Eating  
Philosopher 5 is Hungry  
Philosopher 5 takes fork 4 and 5  
Philosopher 5 is Eating  
Philosopher 3 putting fork 2 and 3 down  
Philosopher 3 is thinking  
Philosopher 2 takes fork 1 and 2  
Philosopher 2 is Eating  
Philosopher 5 putting fork 4 and 5 down  
Philosopher 5 is thinking  
Philosopher 4 takes fork 3 and 4  
Philosopher 4 is Eating  
Philosopher 2 putting fork 1 and 2 down  
Philosopher 2 is thinking  
Philosopher 1 takes fork 5 and 1  
Philosopher 1 is Eating  
Philosopher 3 is Hungry  
Philosopher 5 is Hungry  
Philosopher 4 putting fork 3 and 4 down  
Philosopher 4 is thinking
```

Conclusion:

Hence completed the experiment for Dining philosopher's problem successfully.