

Practical no. 1

Aim: Using Linux Commands.

Theory:

- **ls** - The **ls** command provides a listing of a directory's contents. Its function is similar to the **dir** command in DOS.
The **-l** option generates a listing that shows permissions, owner and group membership, size, modification date, and the name for everything in the directory.
- **pwd** - The **pwd** command shows the current directory (**pwd** stands for “print working directory”). When you first log in to your account, the working directory is **/home/username**. This directory is your home directory, and is also represented by the tilde sign **~**.
- **cd** - To change to another directory, use the **cd** command.
- **find** - The **find** command looks for a file or set of files in a directory and all subdirectories beneath it.
- **mkdir** - To create a directory, use the **mkdir** command.
- **cp** - To copy a file, use the **cp** command. When you copy a file, you specify the source and destination locations.
- **mv** - To move a file, use the **mv** command. As with the **cp** command, you specify the source and destination locations.
- **rm** - To delete a file, use the **rm** command.
- **cat** - This will print out the entire contents of a document file to the terminal.
- **man** - It shows manual page of a command.
- **help** - It shows all commands with their basic info.
- **echo** - The echo command helps us move some data, usually text into a file.
- **sudo** - A widely used command in the Linux command line, sudo stands for "SuperUser Do". So, if you want any command to be done with administrative or root privileges, you can use the sudo command.
- **chmod** - Use chmod to make a file executable and to change the permission granted in Linux.

Practical no. 2

Aim : Writing Shell Scripts.

Theory:

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

Shell Keywords – if, else, break etc.

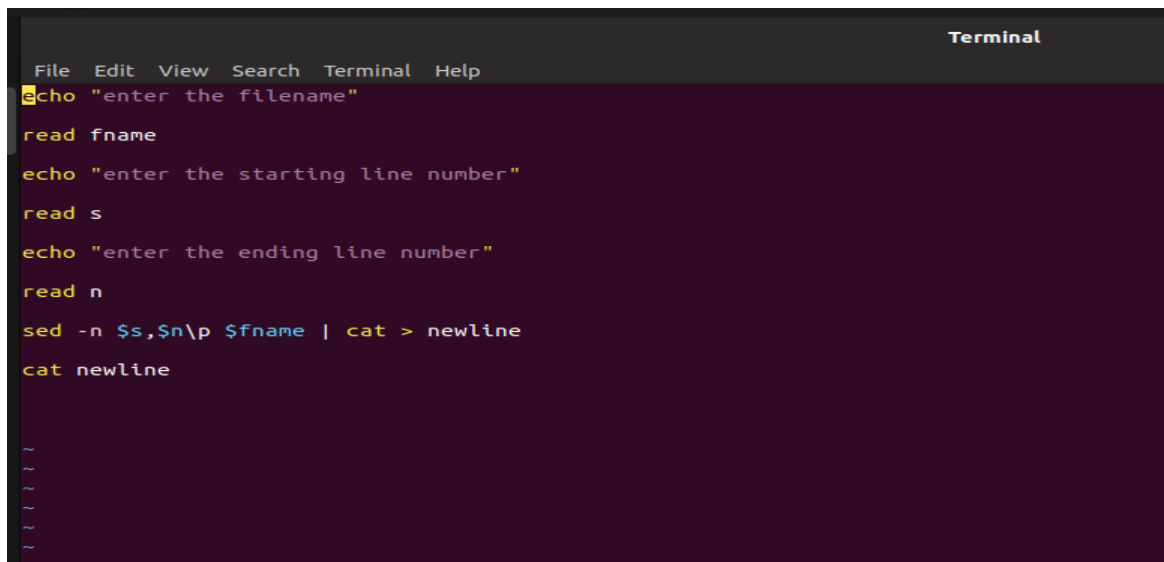
Shell commands – cd, ls, echo, pwd, touch etc.

Functions

Control flow – if..then..else, case and shell loops etc.

Program:

Shell Script:



```
File Edit View Search Terminal Help
echo "enter the filename"
read fname
echo "enter the starting line number"
read s
echo "enter the ending line number"
read n
sed -n $s,$n\p $fname | cat > newline
cat newline

~
~
~
~
~
~
```

Actual File:



The screenshot shows a text editor window titled "Text Editor" with a file named "firstFile.txt" open. The file contains the following text:

```
hello!!!  
Life is a 2017 American sci-fi horror film directed by Daniel Espinosa, written by Rhett Reese and Paul Wernick and starring Jake Gyllenhaal, Rebecca Ferguson, and Ryan Reynolds. The film follows a six-member crew of the International Space Station that uncovers the first evidence of life on Mars.  
  
The first co-production between Skydance Media and Sony Pictures, the film had its world premiere at South by Southwest on March 18, 2017, and was theatrically released in the United States by Columbia Pictures on March 24, 2017. It received mixed reviews, with praise for its acting, visuals and screenplay, but some criticism for lack of originality. The film grossed $100 million worldwide.
```

OUTPUT:



The screenshot shows a terminal window with the following output:

```
enter the filename  
firstFile.txt  
enter the starting line number  
1  
enter the ending line number  
3  
hello!!!  
Life is a 2017 American sci-fi horror film directed by Daniel Espinosa, written by Rhett Reese and Paul Wernick and starring Jake Gyllenhaal, Rebecca Ferguson, and Ryan Reynolds. The film follows a six-member crew of the International Space Station that uncovers the first evidence of life on Mars.
```

Practical no. 3

Aim : To implement FCFS Scheduling.

Theory:

Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.


Program:

```
#include<bits/stdc++.h>
using namespace std;
struct process
{
    int art,bt,wt,tat,id,ct;
};
bool compare(process p1,process p2)
{
    return p1.art<p2.art;
}
void calculate_wt(process p[],int n)
{
    int service_time = 0;
    for(int i=0;i<n;i++)
    {
        p[i].wt = service_time-p[i].art;
        service_time +=p[i].bt;
        p[i].ct=service_time;
        if (p[i].wt < 0)
            p[i].wt = 0;
    }
}
void calculate_tat(process p[],int n)
{
    for(int i=0;i<n;i++)
        p[i].tat = p[i].wt+p[i].bt;
}
int main()
{
    cout<<"enter no. of processes : ";
    int n;
    process p[n];
    cout<<"enter the arrival time and burst time of each process : ";
```

```

    for(int i=0;i<n;i++)
    {
        cin>>p[i].art>>p[i].bt;
        p[i].id = i+1;
    }
    sort(p,p+n,compare);
    calculate_tat(p,n);
    cout<<setw(4)<<"AT"<<setw(4)<<"BT"<<setw(4)<<"CT"<<setw(4)<<"TAT"<<setw(4)<<"
    WT\n";
    for(int i=0;i<n;i++)
    {
        cout<<setw(4)<<p[i].art<<setw(4)<<p[i].bt<<setw(4)<<p[i].ct<<setw(4)<<p[i].tat<<setw(4)
        <<p[i].wt<<"\n";
    }
    float atat=0;
    for(int i=0;i<n;i++)
        atat+=p[i].tat;
    cout<<"average turn around time: "<<atat/n<<"\n";
    float awt;
    for(int i=0;i<n;i++)
        awt+=p[i].wt;
    cout<<"average waiting time: "<<awt/n<<"\n";
}

```

 F:\document\Practice\stack\lab2.exe

```

enter no. of processes : 5
enter the arrival time and burst time of each process : 2 3
0 1
3 6
8 9
4 2
  AT  BT  CT  TAT  WT
   0   1   1   1   0
   2   3   4   3   0
   3   6  10   7   1
   4   2  12   8   6
   8   9  21  13   4
average turn around time: 6.4
average waiting time: 2.2

Process returned 0 (0x0)   execution time : 17.105 s
Press any key to continue.

```

Practical no. 4

Aim: To implement SRN scheduling.

Theory:

It is preemptive mode of SJF algorithm in which we give priority to the process having shortest burst time remaining.

Program :

```
#include <bits/stdc++.h>
using namespace std;
struct Process {
    int pid;
    int bt;
    int art;
};
void findWaitingTime(Process proc[], int n,
                     int wt[])
{
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = proc[i].bt;
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    bool check = false;
    while (complete != n) {

        for (int j = 0; j < n; j++) {
            if ((proc[j].art <= t) &&
                (rt[j] < minm) && rt[j] > 0) {
                minm = rt[j];
                shortest = j;
                check = true;
            }
        }
        if (check == false) {
            t++;
            continue;
        }
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;

        if (rt[shortest] == 0) {

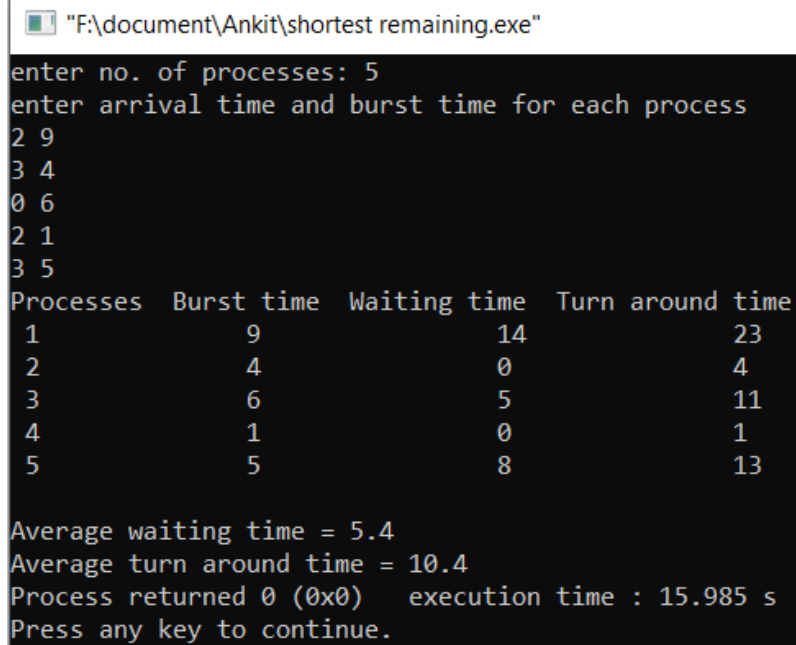
            complete++;
            check = false;
        }
    }
}
```

```

        finish_time = t + 1;
        wt[shortest] = finish_time -
            proc[shortest].bt -
            proc[shortest].art;
        if (wt[shortest] < 0)
            wt[shortest] = 0;
    }
    t++;
}
}
void findTurnAroundTime(Process proc[], int n,
    int wt[], int tat[])
{
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0,
        total_tat = 0;
    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);
    cout << "Processes "
        << " Burst time "
        << " Waiting time "
        << " Turn around time\n";
    for (int i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << proc[i].pid << "\t\t"
            << proc[i].bt << "\t\t " << wt[i]
            << "\t\t " << tat[i] << endl;
    }
    cout << "\nAverage waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}
int main()
{
    int n;
    cout<<"enter no. of processes: ";
    cin>>n;
    Process proc[n];
    cout<<"enter arrival time and burst time for each process\n";
    for(int i=0;i<n;i++)
    {
        proc[i].pid=i+1;
        cin>>proc[i].art>>proc[i].bt;
    }
}

```

```
}  
findavgTime(proc, n);  
return 0;  
}
```

Output:

```
"F:\document\Ankit\shortest remaining.exe"  
enter no. of processes: 5  
enter arrival time and burst time for each process  
2 9  
3 4  
0 6  
2 1  
3 5  
Processes  Burst time  Waiting time  Turn around time  
1          9          14          23  
2          4           0           4  
3          6           5          11  
4          1           0           1  
5          5           8          13  
  
Average waiting time = 5.4  
Average turn around time = 10.4  
Process returned 0 (0x0)   execution time : 15.985 s  
Press any key to continue.
```


Practical no. 5

Aim: To implement RR scheduling.

Theory:

Each process is assigned a fixed time (Time Quantum/Time Slice) in cyclic way. It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Program:

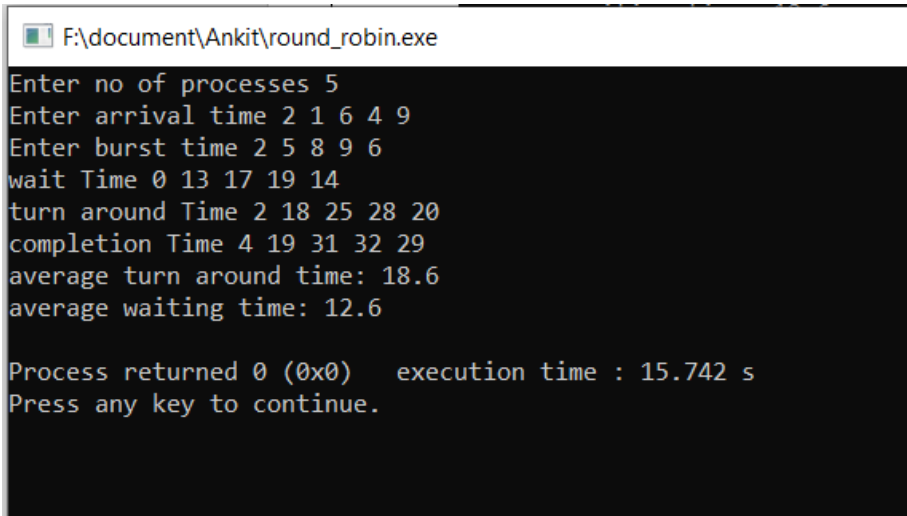
```
#include<iostream>
#include<queue>
using namespace std;
int main()
{
    int n;
    cout<<"Enter no of processes ";
    cin>>n;
    bool completed[n];
    bool inQ[n];
    int arrivalTime[n];
    int burstTime[n],bT[n];
    int completeTime[n]={0};
    int turnAroundTime[n]={0};
    int waitTime[n]={0};
    cout<<"Enter arrival time ";
    for(int i=0;i<n;i++)
    {
        completed[i]=false;
        inQ[i]=false;
        cin>>arrivalTime[i];
    }
    cout<<"Enter burst time ";
    for(int i=0;i<n;i++)
    {
        cin>>burstTime[i];
```

```
    bT[i]=burstTime[i];
}
int tq=2;
int currentTime=arrivalTime[0];
queue<int>Q;
Q.push(0);
inQ[0]=true;
while(!Q.empty())
{
    int job=Q.front();
    Q.pop();
    //process the job
    if(burstTime[job]>tq )
    {
        currentTime+=tq;
        burstTime[job]-=tq;
    }
    else
    {
        currentTime+=burstTime[job];
        completed[job]=true;
        burstTime[job]=0;
    }
    if(burstTime[job]==0)
    {
        inQ[job]=false;
        completed[job]=true;
        completeTime[job]=currentTime;
        turnAroundTime[job]=completeTime[job]-arrivalTime[job];
        waitTime[job]=turnAroundTime[job]-bT[job];
    }
    bool check=false;
    for(int i=0;i<n;i++)
    {
        if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)
        {
            check=true;
            Q.push(i);
            inQ[i]=true;
        }
    }
    if(check==false)
    {
        for(int i=0;i<n;i++)
        {
            if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)
```

```
        {
            currentTime=arrivalTime[i];
            check=true;
            Q.push(i);
            inQ[i]=true;
            break;
        }
    }
}
if(burstTime[job]!=0)
{
    inQ[job]=true;
    Q.push(job);
}
}
cout<<"wait Time ";
for(int i=0;i<n;i++)
{
    cout<<waitTime[i]<<" ";
}cout<<endl;

cout<<"turn around Time ";
for(int i=0;i<n;i++)
{
    cout<<turnAroundTime[i]<<" ";
}cout<<endl;

cout<<"completion Time ";
for(int i=0;i<n;i++)
{
    cout<<completeTime[i]<<" ";
}cout<<endl;
float atat=0;
for(int i=0;i<n;i++)
{
    atat+=turnAroundTime[i];
}
cout<<"average turn around time: "<<atat/n<<"\n";
float awt;
for(int i=0;i<n;i++)
{
    awt+=waitTime[i];
}
cout<<"average waiting time: "<<awt/5<<"\n";
return 0;
}
```

Output:

```
F:\document\Ankit\round_robin.exe
Enter no of processes 5
Enter arrival time 2 1 6 4 9
Enter burst time 2 5 8 9 6
wait Time 0 13 17 19 14
turn around Time 2 18 25 28 20
completion Time 4 19 31 32 29
average turn around time: 18.6
average waiting time: 12.6

Process returned 0 (0x0)   execution time : 15.742 s
Press any key to continue.
```

Practical no. 6

Aim: To implement LCN scheduling.

Theory:

It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

Algorithm:

1. First, sort the processes in increasing order of their Arrival Time.
2. Choose the process having least arrival time but with most Burst Time. Then process it for 1 unit. Check if any other process arrives upto that time of execution or not.
3. Repeat the above both steps until execute all the processes.

Program:

```
#include<iostream>
#include<queue>
using namespace std;
int main()
{
    int n;
    cout<<"Enter no of processes ";
    cin>>n;

    bool completed[n];
    int arrivalTime[n];
    int burstTime[n];
    int bt[n];
    int completeTime[n]={0};
    int turnAroundTime[n]={0};
    int waitTime[n]={0};

    cout<<"Enter arrival time ";
    for(int i=0;i<n;i++)
    {
        completed[i]=false;
        cin>>arrivalTime[i];
    }

    cout<<"Enter burst time ";
    for(int i=0;i<n;i++)
    {
        cin>>burstTime[i];
        bt[i]=burstTime[i];
    }
```

```
int currentTime=arrivalTime[0];
queue<int>Q;
Q.push(0);
while(true)
{
    int job=Q.front();
    Q.pop();
    currentTime++;
    burstTime[job]--;
    //process the job
    if(burstTime[job]==0)
    {
        completed[job]=true;
        completeTime[job]=currentTime;
        turnAroundTime[job]=completeTime[job]-arrivalTime[job];
        waitTime[job]=turnAroundTime[job]-bt[job];
    }
    // find the longest remaining
    int mxIndex=-1,mxBurst=-1;

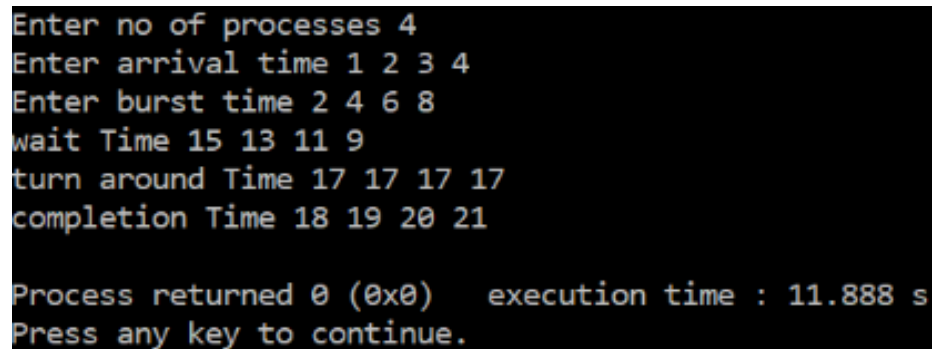
    for(int i=0;i<n;i++)
    {
        if(completed[i]==false && arrivalTime[i]<=currentTime && burstTime[i]>mxBurst)
        {
            mxBurst=burstTime[i];
            mxIndex=i;
        }
    }
    if(mxIndex== -1)
    {
        for(int i=0;i<n;i++)
        {
            if(completed[i]==false)
            {
                currentTime=arrivalTime[i];
                mxIndex=i;
                break;
            }
        }
    }
    if(mxIndex== -1)
    {
        break;
    }
    Q.push(mxIndex);
}
```

```
cout<<"wait Time ";
for(int i=0;i<n;i++)
{
    cout<<waitTime[i]<<" ";
}cout<<endl;

cout<<"turn around Time ";
for(int i=0;i<n;i++)
{
    cout<<turnAroundTime[i]<<" ";
}cout<<endl;

cout<<"completion Time ";
for(int i=0;i<n;i++)
{
    cout<<completeTime[i]<<" ";
}cout<<endl;
return 0;
}
```

Output:

A screenshot of a terminal window with a black background and white text. The text shows the input and output of a program. The input consists of four lines: 'Enter no of processes 4', 'Enter arrival time 1 2 3 4', 'Enter burst time 2 4 6 8', and 'wait Time 15 13 11 9'. The output consists of three lines: 'turn around Time 17 17 17 17', 'completion Time 18 19 20 21', and 'Process returned 0 (0x0) execution time : 11.888 s'. The last line is 'Press any key to continue.' followed by a cursor.

```
Enter no of processes 4
Enter arrival time 1 2 3 4
Enter burst time 2 4 6 8
wait Time 15 13 11 9
turn around Time 17 17 17 17
completion Time 18 19 20 21

Process returned 0 (0x0) execution time : 11.888 s
Press any key to continue.
```

Practical no. 7

Aim: To implement Priority based Non preemptive Scheduling.

Theory :

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with highest priority is to be executed first and so on. Processes with same priority are executed on first come first served basis.

Program:

```
#include <bits/stdc++.h>
using namespace std;
struct process
{
    int at,bt,pr,pno;
};

process proc[50];
bool comp(process a,process b)
{
    if(a.at == b.at)
    {
        return a.pr<b.pr;
    }
    else
    {
        return a.at<b.at;
    }
}

void get_wt_time(int wt[],int n)
{
    int service[50];
    service[0]=0;
    wt[0]=0;
    for(int i=1;i<n;i++)
    {
        service[i]=proc[i-1].bt+service[i-1];

        wt[i]=service[i]-proc[i].at+1;
        if(wt[i]<0)
        {
            wt[i]=0;
        }
    }
}
```



```

    }
    void get_tat_time(int tat[],int wt[],int n)
    {
        for(int i=0;i<n;i++)
        {
            tat[i]=proc[i].bt+wt[i];
        }
    }
    void findgc(int n)
    {
        int wt[50],tat[50];
        double wavg=0,tavg=0;
        get_wt_time(wt,n);
        get_tat_time(tat,wt,n);

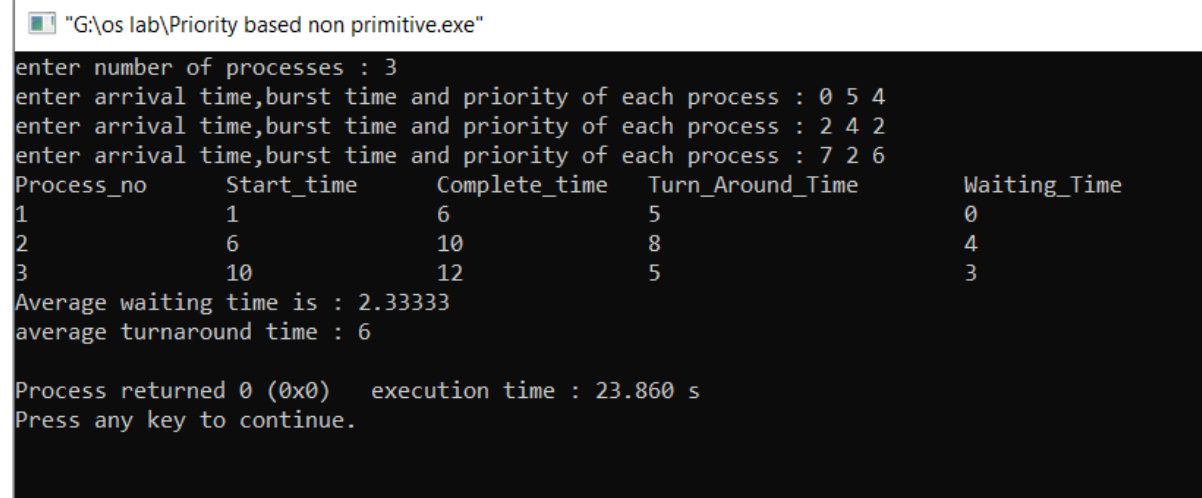
        int stime[50],ctime[50];
        stime[0]=1;
        ctime[0]=stime[0]+tat[0];
        for(int i=1;i<n;i++)
        {
            stime[i]=ctime[i-1];
            ctime[i]=stime[i]+tat[i]-wt[i];
        }
        cout<<"Process_no\tStart_time\tComplete_time\tTurn_Around_Time\tWaiting_Time"
        "<<endl;
        for(int i=0;i<n;i++)
        {
            wavg += wt[i];
            tavg += tat[i];

            cout<<proc[i].pno<<"\t\t"<<
                stime[i]<<"\t\t"<<ctime[i]<<"\t\t"<<
                tat[i]<<"\t\t"<<wt[i]<<endl;
        }
        cout<<"Average waiting time is : ";
        cout<<wavg/(float)n<<endl;
        cout<<"average turnaround time : ";
        cout<<tavg/(float)n<<endl;
    }
    int main()
    {
        int n;
        cout<<"enter number of processes : ";
        cin>>n;
        for(int i=0;i<n;i++)
        {

```

```
        cout<<"enter arrival time,burst time and priority of each process : ";
        cin>>proc[i].at>>proc[i].bt>>proc[i].pr;
        proc[i].pno=i+1;
    }
    sort(proc,proc+n,comp);
    findgc(n);
    return 0;
}
```

Output:



```
"G:\os lab\Priority based non primitive.exe"
enter number of processes : 3
enter arrival time,burst time and priority of each process : 0 5 4
enter arrival time,burst time and priority of each process : 2 4 2
enter arrival time,burst time and priority of each process : 7 2 6
Process_no      Start_time      Complete_time      Turn_Around_Time      Waiting_Time
1                1                6                  5                      0
2                6                10                 8                      4
3               10                12                 5                      3
Average waiting time is : 2.33333
average turnaround time : 6

Process returned 0 (0x0)   execution time : 23.860 s
Press any key to continue.
```

Practical no. 8

Aim: To implement Priority based preemptive Scheduling.

Theory :

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with highest priority is to be executed first and so on. Processes with same priority are executed on first come first served basis.

Program:

```
#include <bits/stdc++.h>
using namespace std;
struct Process {
    int processID;
    int burstTime;
    int tempburstTime;
    int responsetime;
    int arrivalTime;
    int priority;
    int outtime;
    int intime;
};

void insert(Process Heap[], Process value, int* heapsize,
            int* currentTime)
{
    int start = *heapsize, i;
    Heap[*heapsize] = value;
    if (Heap[*heapsize].intime == -1)
        Heap[*heapsize].intime = *currentTime;
    ++(*heapsize);
    while (start != 0 && Heap[(start - 1) / 2].priority >
            Heap[start].priority) {
        Process temp = Heap[(start - 1) / 2];
        Heap[(start - 1) / 2] = Heap[start];
        Heap[start] = temp;
        start = (start - 1) / 2;
    }
}

void order(Process Heap[], int* heapsize, int start)
{
    int smallest = start;
    int left = 2 * start + 1;
    int right = 2 * start + 2;
    if (left < *heapsize && Heap[left].priority <
        Heap[smallest].priority)
```

```

        smallest = left;
    if (right < *heapsize && Heap[right].priority <
        Heap[smallest].priority)
        smallest = right;
    if (smallest != start) {
        Process temp = Heap[smallest];
        Heap[smallest] = Heap[start];
        Heap[start] = temp;
        order(Heap, heapsize, smallest);
    }
}

Process extractminimum(Process Heap[], int* heapsize,
    int* currentTime)
{
    Process min = Heap[0];
    if (min.responsetime == -1)
        min.responsetime = *currentTime - min.arrivalTime;
    --(*heapsize);
    if (*heapsize >= 1) {
        Heap[0] = Heap[*heapsize];
        order(Heap, heapsize, 0);
    }
    return min;
}

bool compare(Process p1, Process p2)
{
    return (p1.arrivalTime < p2.arrivalTime);
}

void scheduling(Process Heap[], Process array[], int n,
    int* heapsize, int* currentTime)
{
    if (heapsize == 0)
        return;

    Process min = extractminimum(Heap, heapsize, currentTime);
    min.outtime = *currentTime + 1;
    --min.burstTime;
    printf("process id = %d current time = %d\n",
        min.processID, *currentTime);
    if (min.burstTime > 0) {
        insert(Heap, min, heapsize, currentTime);
        return;
    }

    for (int i = 0; i < n; i++)
        if (array[i].processID == min.processID) {

```

```

        array[i] = min;
        break;
    }
}
void priority(Process array[], int n)
{
    sort(array, array + n, compare);

    int totalwaitingtime = 0, totalbursttime = 0,
        totalturnaroundtime = 0, i, insertedprocess = 0,
        heapsize = 0, currentTime = array[0].arrivalTime,
        totalresponsetime = 0;

    Process Heap[4 * n];
    for (int i = 0; i < n; i++) {
        totalbursttime += array[i].burstTime;
        array[i].tempburstTime = array[i].burstTime;
    }

    do {
        if (insertedprocess != n) {
            for (i = 0; i < n; i++) {
                if (array[i].arrivalTime == currentTime) {
                    ++insertedprocess;
                    array[i].intime = -1;
                    array[i].responsetime = -1;
                    insert(Heap, array[i], &heapsize, &currentTime);
                }
            }
        }
        scheduling(Heap, array, n, &heapsize, &currentTime);
        ++currentTime;
        if (heapsize == 0 && insertedprocess == n)
            break;
    } while (1);

    for (int i = 0; i < n; i++) {
        totalresponsetime += array[i].responsetime;
        totalwaitingtime += (array[i].outtime - array[i].intime -
            array[i].tempburstTime);
        totalbursttime += array[i].burstTime;
    }
    printf("Average waiting time = %f\n",
        ((float)totalwaitingtime / (float)n));
    printf("Average response time = %f\n",

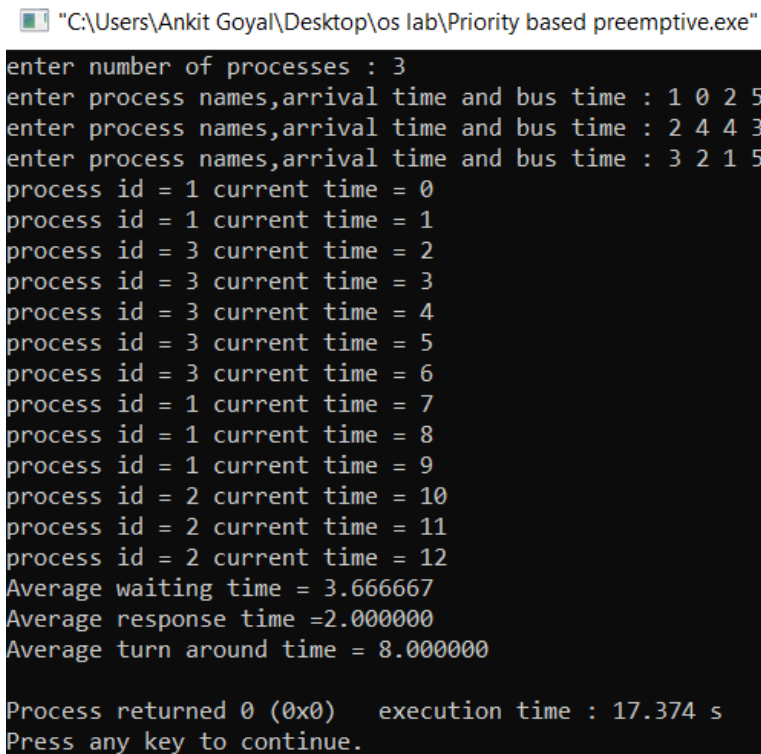
```

```

        ((float)totalresponsetime / (float)n));
    printf("Average turn around time = %f\n",
        ((float)(totalwaitingtime + totalbursttime) / (float)n));
}
int main()
{
    int n,i;
    cout<<"enter number of processes : ";
    cin>>n;
    Process a[n];
    for(int i=0;i<n;i++)
    {
        cout<<"enter process names,arrival time and bus time : ";
        cin>>a[i].processID>>a[i].arrivalTime>>a[i].priority>>a[i].burstTime;
    }
    priority(a, 3);
    return 0;
}

```

Output:



```

"C:\Users\Ankit Goyal\Desktop\os lab\Priority based preemptive.exe"
enter number of processes : 3
enter process names,arrival time and bus time : 1 0 2 5
enter process names,arrival time and bus time : 2 4 4 3
enter process names,arrival time and bus time : 3 2 1 5
process id = 1 current time = 0
process id = 1 current time = 1
process id = 3 current time = 2
process id = 3 current time = 3
process id = 3 current time = 4
process id = 3 current time = 5
process id = 3 current time = 6
process id = 1 current time = 7
process id = 1 current time = 8
process id = 1 current time = 9
process id = 2 current time = 10
process id = 2 current time = 11
process id = 2 current time = 12
Average waiting time = 3.666667
Average response time =2.000000
Average turn around time = 8.000000

Process returned 0 (0x0)   execution time : 17.374 s
Press any key to continue.

```

Practical no. 9

Aim: To implement Bankers Algorithm for deadlock Prevention.

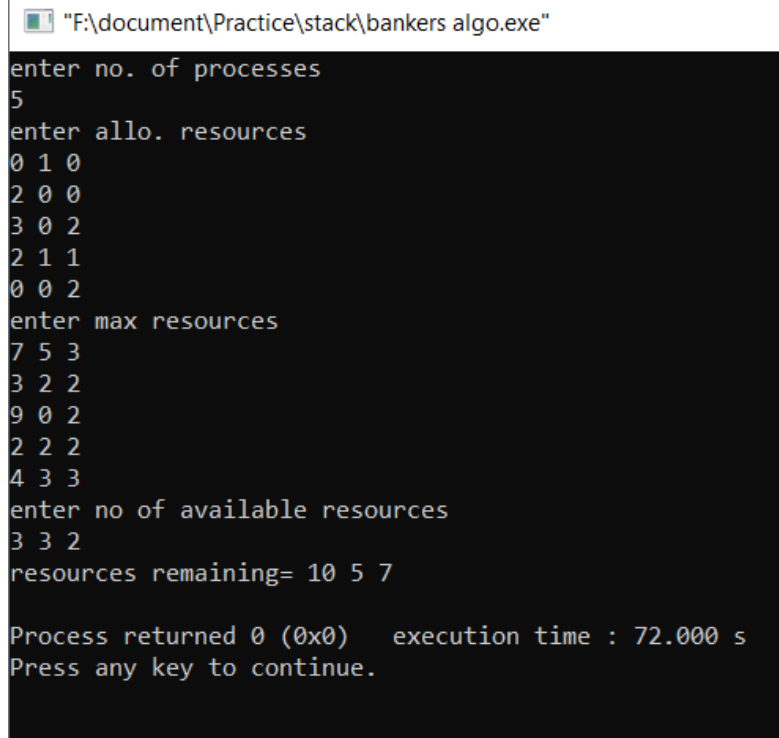
Theory: -

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Program:

```
#include<iostream>
using namespace std;
struct node
{
    int a,b,c;
};
int main()
{
    int n;
    cout<<"enter no. of processes\n";
    cin>>n;
    node allo[n],max[n],need[n];
    node avail;
    cout<<"enter allo. resources\n";
    int i=0,j;
    while(i<n)
    {
        cin>>allo[i].a>>allo[i].b>>allo[i].c;
        i++;
    }
    cout<<"enter max resources\n";
    i=0;
    while(i<n)
    {
        cin>>max[i].a>>max[i].b>>max[i].c;
        i++;
    }
    cout<<"enter no of available resources\n";
    cin>>avail.a>>avail.b>>avail.c;
    i=0;
    while(i<n)
    {
```

```
        need[i].a=max[i].a-allo[i].a;
        need[i].b=max[i].b-allo[i].b;
        need[i].c=max[i].c-allo[i].c;
        i++;
    }
    int flag[n],check=0;
    for(i=0;i<n;i++)
        flag[i]=0;
    i=0;
    while(i<n)
    {
        j=0;
        check=1;
        while(j<n)
        {
            if(flag[j]==0&&( need[j].a<=avail.a && need[j].b<=avail.b && need[j].c<=avail.c ))
            {
                check=0;
                avail.a+=allo[j].a;
                avail.b+=allo[j].b;
                avail.c+=allo[j].c;
                flag[j]=1;
                break;
            }
            j++;
        }
        if(check)
        {
            cout<<"safe seq not possible\n";
            break;
        }
        i++;
    }
    if(!check)
    {
        cout<<"resources remaining= "<<avail.a<<" "<<avail.b<<" "<<avail.c<<"\n";
    }
}
```


Output:

```
"F:\document\Practice\stack\bankers algo.exe"
enter no. of processes
5
enter allo. resources
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter max resources
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
enter no of available resources
3 3 2
resources remaining= 10 5 7

Process returned 0 (0x0)   execution time : 72.000 s
Press any key to continue.
```

Practical no. 10

Aim: To implement Producer Consumer Problem with Bounded Buffer.

Theory: -

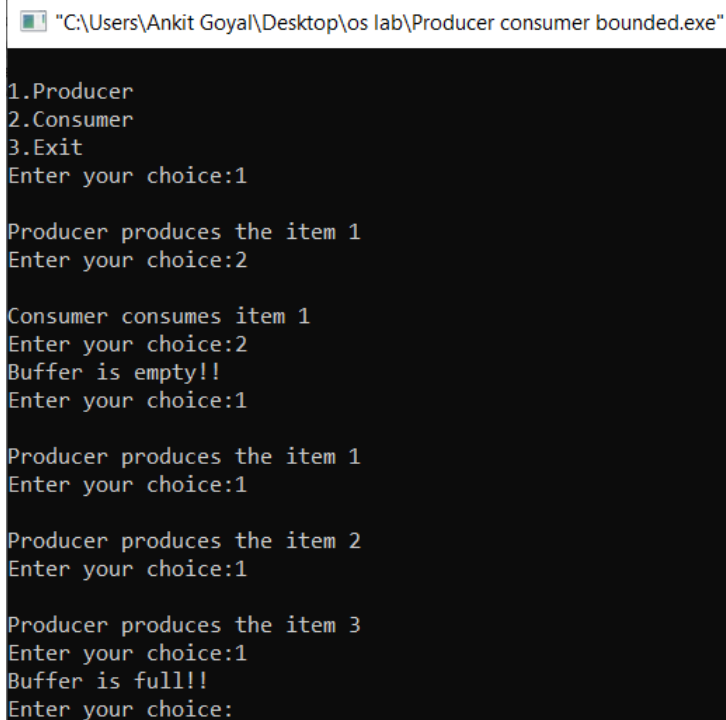
We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

Program:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter 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;
        }
    }
    return 0;
}
```

```
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("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}
```

Output:



```
"C:\Users\Ankit Goyal\Desktop\os lab\Producer consumer bounded.exe"

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:2

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

Practical no. 11

Aim: To implement Producer Consumer Problem with Unbounded Buffer.

Theory: -

We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section

Program:

```
class Q
{
    int n;
    boolean valueSet = false;
    synchronized int get()
    {
        while(!valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
    synchronized void put(int n) {
        while(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}

class Producer implements Runnable
```

```
{
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}
```

Output:

```
<terminated> Temp [Java Application] C:\Program Files\Java\jdk1
Press Control-C to stop.
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
```

Practical no. 12

Aim: To implement Reader Writer problem.

Theory:

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Program :

```
import java.util.Scanner;
class Database
{
    private int readers;
    public Database()
    {
        this.readers = 0;
    }
    public void read(int number)
    {
        synchronized(this)
        {
            this.readers++;
            System.out.println("Reader " + number + " starts reading.");
        }
        final int DELAY = 5000;
        try
        {
            Thread.sleep((int) (Math.random() * DELAY));
        }
        catch (InterruptedException e) {}
        synchronized(this)
        {
            System.out.println("Reader " + number + " stops reading.");
            this.readers--;
            if (this.readers == 0)
            {
                this.notifyAll();
            }
        }
    }
}
```

```
public synchronized void write(int number)
{
    while (this.readers != 0)
    {
        try
        {
            this.wait();
        }
        catch (InterruptedException e) {}
    }
    System.out.println("Writer " + number + " starts writing.");
    final int DELAY = 5000;
    try
    {
        Thread.sleep((int) (Math.random() * DELAY));
    }
    catch (InterruptedException e) {}
    System.out.println("Writer " + number + " stops writing.");
    this.notifyAll();
}
}
class Reader extends Thread
{
    private static int readers = 0;
    private int number;
    private Database database;
    public Reader(Database database)
    {
        this.database = database;
        this.number = Reader.readers++;
    }
    public void run()
    {
        while (true)
        {
            final int DELAY = 5000;
            try
            {
                Thread.sleep((int) (Math.random() * DELAY));
            }
            catch (InterruptedException e) {}
        }
    }
}
```

```
        this.database.read(this.number);
    }
}
class Writer extends Thread
{
    private static int writers = 0;

    private int number;
    private Database database;
    public Writer(Database database)
    {
        this.database = database;
        this.number = Writer.writers++;
    }
    public void run()
    {
        while (true)
        {
            final int DELAY = 5000;
            try
            {
                Thread.sleep((int) (Math.random() * DELAY));
            }
            catch (InterruptedException e) {}
            this.database.write(this.number);
        }
    }
}
public class ReaderWriterProblem
{
    public static void main(String[] arg)
    {
        int r , w;
        System.out.println("Enter the number of readers and writers");
        Scanner in = new Scanner(System.in);
        r = in.nextInt();
        w = in.nextInt();
        final int READERS = r;
        final int WRITERS = w;
        Database database = new Database();
```



```
    for (int i = 0; i < READERS; i++)
    {
        new Reader(database).start();
    }
    for (int i = 0; i < WRITERS; i++)
    {
        new Writer(database).start();
    }
}
```

Output:

```
<terminated> Temp [Java Application] C:\Program Files\Java\jdk1.8.0
Enter the number of readers and writers
3 3
Writer 0 starts writing.
Writer 0 stops writing.
Writer 1 starts writing.
Writer 1 stops writing.
Reader 2 starts reading.
Reader 1 starts reading.
```

Practical no. 13

Aim: To implement Dining Philosopher 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 pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Program:

```
#include<iostream>
#define n 2
using namespace std;
int cPhil = 0,i;
struct fork{
int taken;
}ForkAvil[n];
struct ph{
int left;
int right;
}phs[n];

void Dine(int philID){
    if(phs[philID].left==10 && phs[philID].right==10)
        cout<<"Philosopher "<<philID+1<<" completed his dinner\n";
    else if(phs[philID].left==1 && phs[philID].right==1){
        cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

        phs[philID].left = phs[philID].right = 10;
        int otherFork = philID-1;

        if(otherFork== -1)
            otherFork=(n-1);

        ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;
        cout<<"Philosopher "<<philID+1<<" released fork "<<philID+1<<" and fork
"<<otherFork+1<<"\n";
        cPhil++;
    }

    else if(phs[philID].left==1 && phs[philID].right==0){
        if(philID==(n-1)){
            if(ForkAvil[philID].taken==0){
                ForkAvil[philID].taken = phs[philID].right = 1;
```

```

        cout<<"Fork "<<philID+1<<" taken by philosopher
"<<philID+1<<"\n";
    }else{
        cout<<"Philosopher "<<philID+1<<" is waiting for fork
"<<philID+1<<"\n";
    }
}else
{
    int dupphilID = philID;
    philID-=1;

    if(philID== -1)
        philID=(n-1);


    if(ForkAvil[philID].taken == 0){
        ForkAvil[philID].taken = phs[dupphilID].right = 1;
        cout<<"Fork "<<philID+1<<" taken by Philosopher
"<<dupphilID+1<<"\n";
    }
else{
        cout<<"Philosopher "<<dupphilID+1<<" is waiting for Fork
"<<philID+1<<"\n";
    }
}
}
else if(phs[philID].left==0){
    if(philID==(n-1)){
        if(ForkAvil[philID-1].taken==0){
            ForkAvil[philID-1].taken = phs[philID].left = 1;
            cout<<"Fork "<<philID<<" taken by philosopher
"<<philID+1<<"\n";
        }else{
            cout<<"Philosopher "<<philID+1<<" is waiting for fork
"<<philID<<"\n";
        }
    }
else
{
    if(ForkAvil[philID].taken == 0){
        ForkAvil[philID].taken = phs[philID].left = 1;
        cout<<"Fork "<<philID+1<<" taken by Philosopher
"<<philID+1<<"\n";
    }else{
        cout<<"Philosopher "<<philID+1<<" is waiting for Fork
"<<philID+1<<"\n";
    }
}
}

```

```
        }
    }
}
int main(){
for(i=0;i<n;i++){
    ForkAvil[i].taken=phs[i].left=phs[i].right=0;
    while(cPhil<n){
        for(i=0;i<n;i++)    Dine(i);
        cout<<"\nTill now num of philosophers completed dinner are
"<<cPhil<<"\n\n";
    }

    return 0;
}
```

Output:

 "C:\Users\Ankit Goyal\Desktop\os lab\Dining phiolosopher.exe"

```
Fork 1 taken by Philosopher 1
Philosopher 2 is waiting for fork 1

Till now num of philosophers completed dinner are 0

Fork 2 taken by Philosopher 1
Philosopher 2 is waiting for fork 1

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 2
Fork 1 taken by philosopher 2

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Fork 2 taken by philosopher 2

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1

Till now num of philosophers completed dinner are 2
```

Practical no. 14

Aim: To implement Reader Writer Problem using Semaphores.

Theory:

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Program:

```
#include<semaphore.h>
#include<stdio.h>
#include<pthread.h>
# include<bits/stdc++.h>
using namespace std;
void *reader(void *);
void *writer(void *);
int readcount=0,writecount=0,sh_var=5,bsize[5];
sem_t x,y,z,rsem,wsem;
pthread_t r[3],w[2];
void *reader(void *i)
{
    cout << "\n-----";
    cout << "\n\n reader-" << i << " is reading";
    sem_wait(&z);
    sem_wait(&rsem);
    sem_wait(&x);
    readcount++;
    if(readcount==1)
        sem_wait(&wsem);
    sem_post(&x);
    sem_post(&rsem);
    sem_post(&z);
    cout << "\nupdated value : " << sh_var;
    sem_wait(&x);
    readcount--;
    if(readcount==0)
        sem_post(&wsem);
```

```
        sem_post(&x);
    }
void *writer(void *i)
{
    cout << "\n\n writer-" << i << "is writing";
    sem_wait(&y);
    writecount++;
    if(writecount==1)
        sem_wait(&rsem);
    sem_post(&y);
    sem_wait(&wsem);
    sh_var=sh_var+5;
    sem_post(&wsem);
    sem_wait(&y);
    writecount--;
    if(writecount==0)
        sem_post(&rsem);
    sem_post(&y);}
int main()
{
    sem_init(&x,0,1);
    sem_init(&wsem,0,1);
    sem_init(&y,0,1);
    sem_init(&z,0,1);
    sem_init(&rsem,0,1);
    pthread_create(&r[0],NULL,(void *)reader,(void *)0);
    pthread_create(&w[0],NULL,(void *)writer,(void *)0);
    pthread_create(&r[1],NULL,(void *)reader,(void *)1);
    pthread_create(&r[2],NULL,(void *)reader,(void *)2);
    pthread_create(&r[3],NULL,(void *)reader,(void *)3);
    pthread_create(&w[1],NULL,(void *)writer,(void *)3);
    pthread_create(&r[4],NULL,(void *)reader,(void *)4);
    pthread_join(r[0],NULL);
    pthread_join(w[0],NULL);
    pthread_join(r[1],NULL);
    pthread_join(r[2],NULL);
    pthread_join(r[3],NULL);
```

```
pthread_join(w[1],NULL);  
pthread_join(r[4],NULL);  
}
```

Output:

```
-----  
reader-0 is reading  
updated value : 5  
  
writer-0 is writing  
-----  
reader-1 is reading  
updated value : 10  
-----  
reader-2 is reading  
updated value : 10  
-----  
reader-3 is reading  
updated value : 10  
  
writer-3 is writing  
-----  
reader-4 is reading
```

Practical no. 15

Aim: To implement Dining Philosopher Problem using Semaphores.

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 pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Program:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <windows.h>
#include <unistd.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[phnum] = EATING;

        usleep(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]);
    }
}

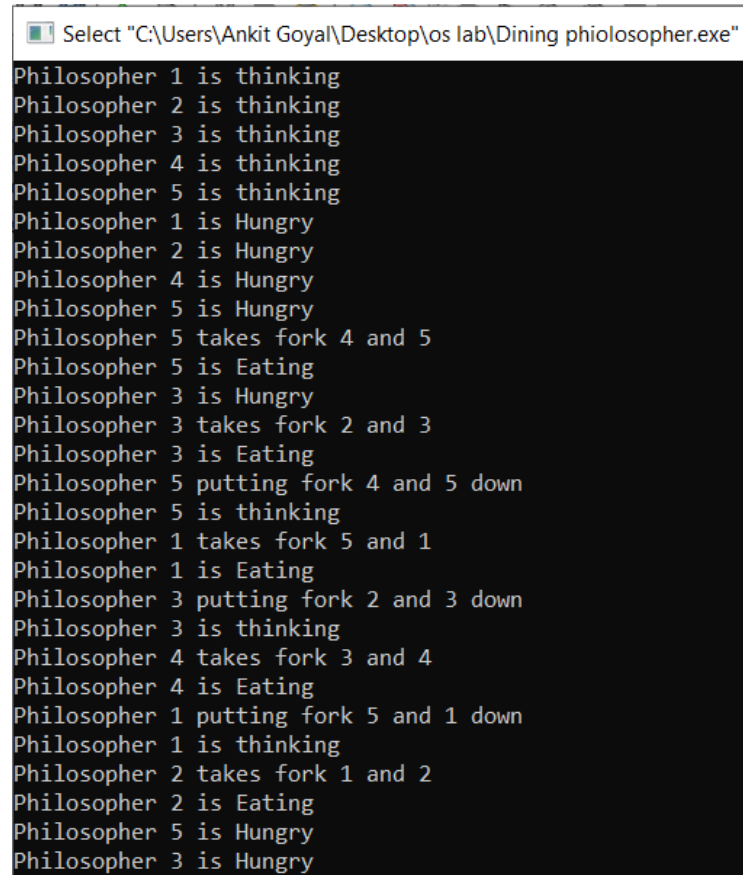
void take_fork(int phnum)
{
    sem_wait(&mutex);
```



```
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    usleep(1);
}
void put_fork(int phnum)
{
    sem_wait(&mutex);
    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* philospher(void* num)
{
    while (1) {
        int* i = (int *)num;
        usleep(1);
        take_fork(*i);
        usleep(0);
        put_fork(*i);
    }
}

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

Output:



```
Select "C:\Users\Ankit Goyal\Desktop\os lab\Dining phiolosopher.exe"
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 is Hungry
Philosopher 3 is Hungry
```

Practical no. 16

Aim: To implement Dekker's algorithm.

Theory:

Dekker's algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

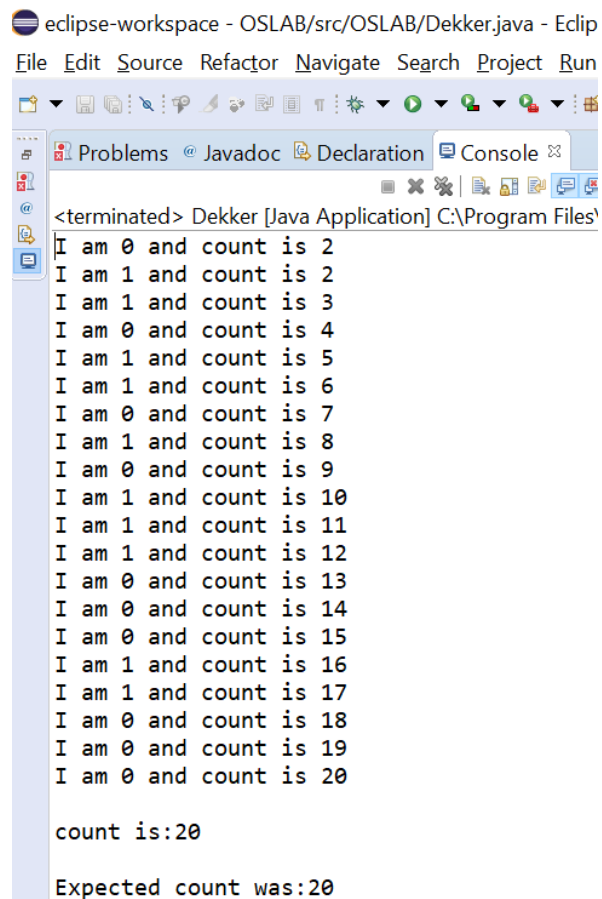
Program:

```
public class Dekker extends Thread{

    public int thread_id;
    public static volatile int turn;
    public static final int countToThis=10;
    public static final int numberOfThreads=2;
    public static volatile int count=0;
    public static volatile boolean[] flag=new boolean[numberOfThreads];
    public Dekker(int id){
        thread_id=id;
    }
    public void run(){
        int scale=10;
        for(int i=0;i<countToThis;++i){
            count++;
            flag[this.thread_id]=true;
            while(flag[1-this.thread_id]){
                if(turn==1-thread_id){
                    flag[this.thread_id]=false;
                    while(turn==1-thread_id){ }
                    flag[this.thread_id]=true;
                }
            }
            System.out.println("I am "+thread_id+" and count is "+count);
            turn=1-thread_id;
            flag[this.thread_id]=false;
            try{
                sleep((int) (Math.random() * scale));
            }
            catch (InterruptedException e){
            }
        }
    }
    public static void main(String[] args){
        Dekker[] threads=new Dekker[numberOfThreads];
        for(int i=0;i<threads.length;++i){
            threads[i]=new Dekker(i);
            threads[i].start();
        }
    }
}
```

```
}
for(int i=0;i<threads.length;++i){
    try{
        threads[i].join();
    }
    catch (InterruptedException e){
        e.printStackTrace();
    }
}
System.out.println("\ncount is:"+count);
System.out.println("\nExpected count was:"+(countToThis*numberOfThreads));
}
}
```

Output:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The title bar reads 'eclipse-workspace - OSLAB/src/OSLAB/Dekker.java - Eclip'. The menu bar includes 'File', 'Edit', 'Source', 'Refactor', 'Navigate', 'Search', 'Project', and 'Run'. The toolbar contains various icons for file operations and debugging. The console output shows the execution of a Java application named 'Dekker [Java Application] C:\Program Files\'. The output consists of 20 lines of 'I am 0 and count is' followed by a number from 2 to 20, and then 'count is:20' and 'Expected count was:20'.

```
<terminated> Dekker [Java Application] C:\Program Files\
I am 0 and count is 2
I am 1 and count is 2
I am 1 and count is 3
I am 0 and count is 4
I am 1 and count is 5
I am 1 and count is 6
I am 0 and count is 7
I am 1 and count is 8
I am 0 and count is 9
I am 1 and count is 10
I am 1 and count is 11
I am 1 and count is 12
I am 0 and count is 13
I am 0 and count is 14
I am 0 and count is 15
I am 1 and count is 16
I am 1 and count is 17
I am 0 and count is 18
I am 0 and count is 19
I am 0 and count is 20

count is:20

Expected count was:20
```

Practical no. 17

Aim: To implement Bakery algorithm.

Theory:-

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Bakery Algorithm is a critical section solution for N processes. The algorithm preserves the first come first serve property.

Program:

package OSLAB;

```
public class Bakery extends Thread {
    public int thread_id;
    public static final int countToThis = 4;
    public static final int numberOfThreads = 5;
    public static volatile int count = 0;
    private static volatile boolean[] choosing = new boolean[numberOfThreads];
    private static volatile int[] ticket = new int[numberOfThreads];
    public Bakery(int id) {
        thread_id = id;
    }
    public void run() {
        int scale = 2;
        for (int i = 0; i < countToThis; i++) {
            lock(thread_id);
            count = count + 1;
            System.out.println("I am " + thread_id + " and count is: " + count);
            try {
                sleep((int) (Math.random() * scale));
            } catch (InterruptedException e) { }

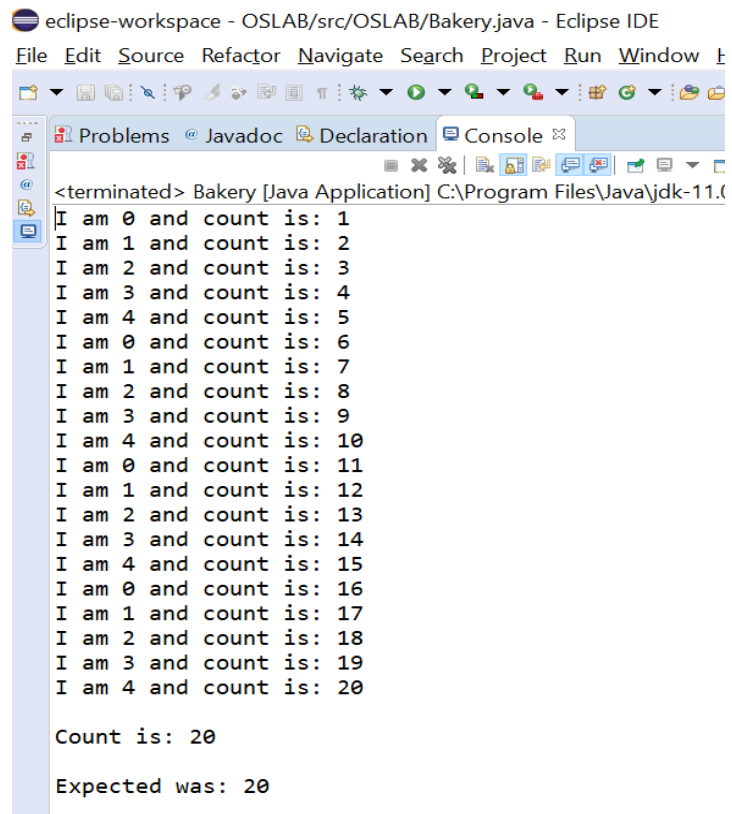
            unlock(thread_id);
        }
    }
    public void lock(int id) {
        choosing[id] = true;
        ticket[id] = findMax() + 1;
        choosing[id] = false;
        for (int j = 0; j < numberOfThreads; j++) {
            if (j == id)
                continue;
            while (choosing[j]) { }
            while (ticket[j] != 0 && (ticket[id] > ticket[j] || (ticket[id] == ticket[j] && id > j))) {
                /* nothing */
            }
        }
    }
    private void unlock(int id) {
        ticket[id] = 0;
    }
}
```

```
}
private int findMax() {
    int m = ticket[0];
    for (int i = 1; i < ticket.length; i++) {
        if (ticket[i] > m)
            m = ticket[i];
    }
    return m;
}
public static void main(String[] args) {

    for (int i = 0; i < numberOfThreads; i++) {
        choosing[i] = false;
        ticket[i] = 0;
    }
    Bakery[] threads = new Bakery[numberOfThreads];

    for (int i = 0; i < threads.length; i++) {
        threads[i] = new Bakery(i);
        threads[i].start();
    }
    for (int i = 0; i < threads.length; i++) {
        try {
            threads[i].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("\nCount is: " + count);
    System.out.println("\nExpected was: " + (countToThis * numberOfThreads));
}
}
```

Output:



The screenshot shows the Eclipse IDE interface with the console window open. The title bar reads 'eclipse-workspace - OSLAB/src/OSLAB/Bakery.java - Eclipse IDE'. The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, and Window. The toolbar contains various icons for file operations and development tools. The console window has tabs for Problems, Javadoc, Declaration, and Console. The console output shows the execution of a Java application named 'Bakery'. It prints a series of lines: 'I am 0 and count is: 1' through 'I am 4 and count is: 20'. After these lines, it prints 'Count is: 20' and 'Expected was: 20'. The application has terminated, as indicated by the '<terminated>' message at the top of the console output.

```
<terminated> Bakery [Java Application] C:\Program Files\Java\jdk-11.0
I am 0 and count is: 1
I am 1 and count is: 2
I am 2 and count is: 3
I am 3 and count is: 4
I am 4 and count is: 5
I am 0 and count is: 6
I am 1 and count is: 7
I am 2 and count is: 8
I am 3 and count is: 9
I am 4 and count is: 10
I am 0 and count is: 11
I am 1 and count is: 12
I am 2 and count is: 13
I am 3 and count is: 14
I am 4 and count is: 15
I am 0 and count is: 16
I am 1 and count is: 17
I am 2 and count is: 18
I am 3 and count is: 19
I am 4 and count is: 20

Count is: 20

Expected was: 20
```

Practical no. 18

Aim: To implement FIFO Page Replacement Policy.

Theory:-

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Program:


```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity)
{
    queue<int> indexes;
    int page_faults = 0;
    set<int> s;
    for (int i=0; i<n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i])==s.end())
            {
                s.insert(pages[i]);
                page_faults++;
                indexes.push(pages[i]);
            }
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int val = indexes.front();
                indexes.pop();
                s.erase(val);
                s.insert(pages[i]);
                indexes.push(pages[i]);
                page_faults++;
            }
        }
    }

    return page_faults;
}

int main()
{
```



```
int n;
cout<<"enter no. of pages: ";
cin>>n;
    int pages[n];
    cout<<"enter page numbers\n";
    for(int i=0;i<n;i++)
        cin>>pages[i];
    int capacity;
    cout<<"enter capacity: ";
    cin>>capacity;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

Output: "H:\os lab\fifo.exe"

```
enter no. of pages: 8
enter page numbers
7 0 1 2 0 3 4 0
enter capacity: 3
7
Process returned 0 (0x0)   execution time : 631.150 s
Press any key to continue.
```

Practical no. 19

Aim: To implement LRU Page Replacement Policy.

Theory:-

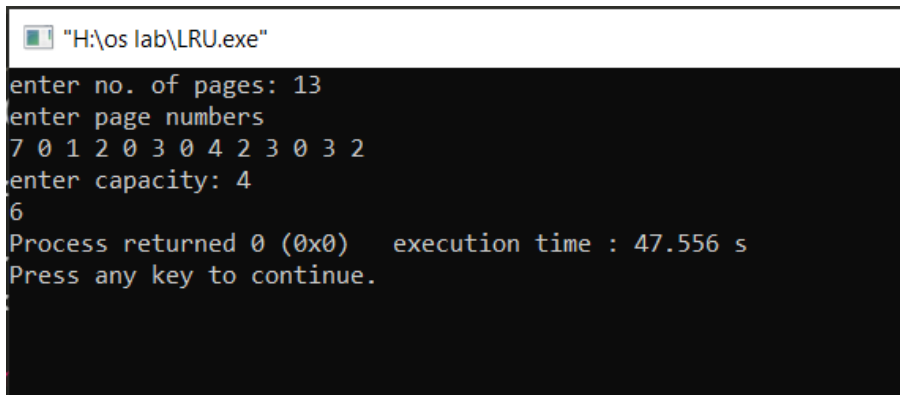
In **Least Recently Used (LRU)** algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely.

Program:

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity)
{
    set<int> s;
    map<int, int> indexes;
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i])==s.end())
            {
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int lru = INT_MAX, val;
                set<int>::iterator it;
                for ( it=s.begin(); it!=s.end(); it++)
                {
                    if (indexes[*it] < lru)
                    {
                        lru = indexes[*it];
                        val = *it;
                    }
                }
                s.erase(val);
                s.insert(pages[i]);
                page_faults++;
            }
        }
    }
}
```

```
        }
        indexes[pages[i]] = i;
    }
}
return page_faults;
}

int main()
{
    int n;
    cout<<"enter no. of pages: ";
    cin>>n;
    int pages[n];
    cout<<"enter page numbers\n";
    for(int i=0;i<n;i++)
    cin>>pages[i];
    int capacity;
    cout<<"enter capacity: ";
    cin>>capacity;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

Output:

```
"H:\os lab\LRU.exe"
enter no. of pages: 13
enter page numbers
7 0 1 2 0 3 0 4 2 3 0 3 2
enter capacity: 4
6
Process returned 0 (0x0)   execution time : 47.556 s
Press any key to continue.
```

Practical no. 20

Aim: To implement Optimal Page Replacement Policy.

Theory:-

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page.

Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

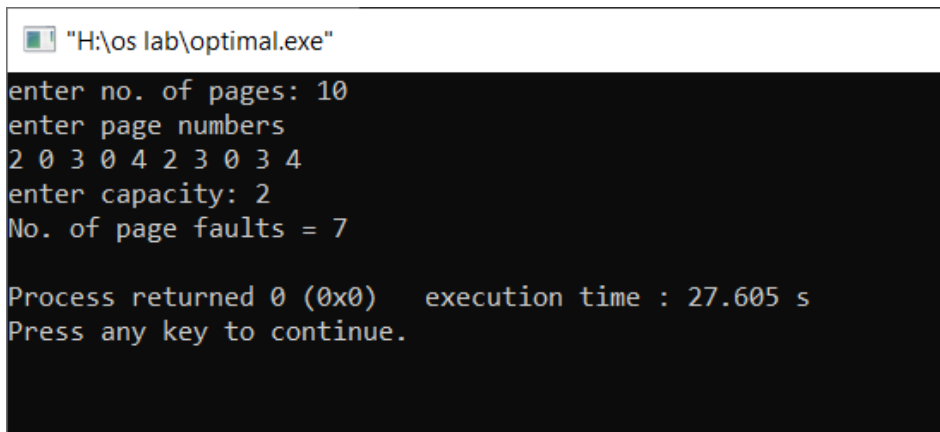
Program:

```
#include <bits/stdc++.h>
using namespace std;
bool search(int key, vector<int>& fr)
{
    for (int i = 0; i < fr.size(); i++)
        if (fr[i] == key)
            return true;
    return false;
}
int predict(int pg[], vector<int>& fr, int pn, int index)
{
    int res = -1, farthest = index;
    for (int i = 0; i < fr.size(); i++) {
        int j;
        for (j = index; j < pn; j++) {
            if (fr[i] == pg[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
            }
        }
        break;
    }
    if (j == pn)
        return i;
}
return (res == -1) ? 0 : res;
}

void optimalPage(int pg[], int pn, int fn)
{
    vector<int> fr;
    int hit = 0;
```

```
        for (int i = 0; i < pn; i++) {
            if (search(pg[i], fr)) {
                hit++;
                continue;
            }
            if (fr.size() < fn)
                fr.push_back(pg[i]);
            else {
                int j = predict(pg, fr, pn, i + 1);
                fr[j] = pg[i];
            }
        }
        cout << "No. of page faults = " << pn - hit << endl;
    }
int main()
{
    int n;
    cout<<"enter no. of pages: ";
    cin>>n;
    int pages[n];
    cout<<"enter page numbers\n";
    for(int i=0;i<n;i++)
    cin>>pages[i];
    int capacity;
    cout<<"enter capacity: ";
    cin>>capacity;
    optimalPage(pages, n, capacity);
    return 0;
}
```

Output:



```
"H:\os lab\optimal.exe"
enter no. of pages: 10
enter page numbers
2 0 3 0 4 2 3 0 3 4
enter capacity: 2
No. of page faults = 7

Process returned 0 (0x0)   execution time : 27.605 s
Press any key to continue.
```