DELHI TECHNOLOGIAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CS207: OPERATING SYSTEMS DESIGN

LAB FILE

SUBMITTED TO:

DR PRASHANT GIRIDHAR SHAMBHARKAR,

ASSISTANT PROFESSOR,

CSE DEPARTMENT

SUBMITTED BY:

KSHITIJ SINHA,

23/CS/225,

A4, G1

INDEX

Experiment 1

AIM

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithms to find turnaround time and waiting time for a given problem. a) FCFS, b) SJF

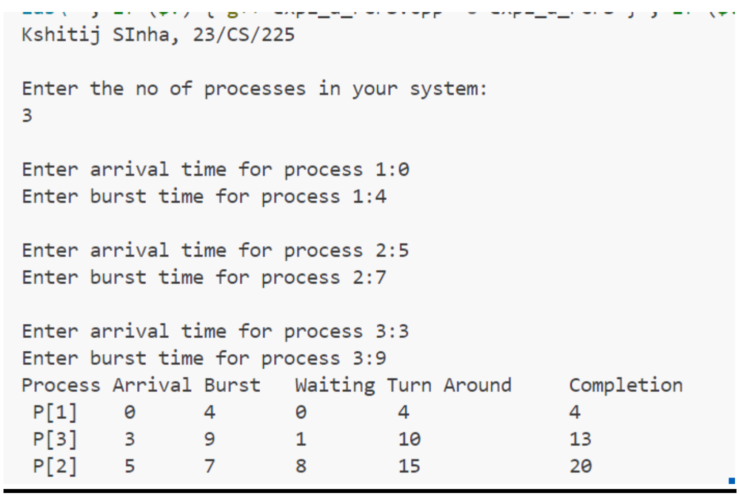
THEORY:

Non-pre-emptive CPU scheduling algorithms are those which cannot stop a program before it finishes executing. In FCFS, the processes are allocated the resources as they come, according to their arrival time. While in SJF, the processes are allocated the resources by deciding which process has the least burst time in the ready queue.

CODE(FCFS):

```
#include<bits/stdc++.h>
using namespace std;
class Process{
    int id, bt, at, ct, tat, wt;
    public:
    void input(Process*,int );
    void calc(Process *,int);
    void show(Process*,int);
    void sort(Process *, int);
};

void Process::input(Process *p,int n){
    for(int i = 0;i<n;i++){
        cout<<"\nEnter arrival time for process "<<i+1<<":";
        cin>>p[i].at;
        cout<<"Enter burst time for process "<<i+1<<":";
        cin>>p[i].bt;
        p[i].id = i+1;
    }
}

void Process::calc(Process*p, int n){
    int sum = 0;
    sum = sum + p[0].at;
    for(int i = 0;i<n;i++){
        sum = sum + p[i].bt;
        p[i].ct = sum;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        if(sum<p[i+1].at){
            int t = p[i+1].at-sum;
            sum = sum+t;
        }
    }
}

void Process::sort(Process*p, int n){
    for(int i=0;i<n-1;i++){
        for(int j=0;j<n-i-1;j++){
            if(p[j].at>p[j+1].at){
                int temp;
                temp = p[j].bt;
                p[j].bt = p[j+1].bt;
                p[j+1].bt = temp;

                temp = p[j].at;
                p[j].at = p[j+1].at;
                p[j+1].at = temp;

                temp = p[j].id;
                p[j].id = p[j+1].id;
                p[j+1].id = temp;
            }
        }
    }
}
void Process::show(Process*p, int n){
    cout<<"Process\tArrival\tBurst\tWaiting\tTurn Around\tCompletion\n";
    for(int i =0;i<n;i++){
        cout<<" P["<<p[i].id<<"]\t "<<p[i].at<<"\t"<<p[i].bt<<"\t"<<p[i].wt<<"\t
"<<p[i].tat<<"\t\t"<<p[i].ct<<"\n";
    }
}
int main(){
    cout<<"Kshitij SInha, 23/CS/225"<<endl;
    int n;
    cout<<"\nEnter the no of processes in your system:\n";
    cin>>n;
    Process *p = new Process[n];
    Process f;
    f.input(p,n);
    f.sort(p, n);
    f.calc(p,n);
    f.show(p,n);
    return 0;
}
```

OUTPUT:



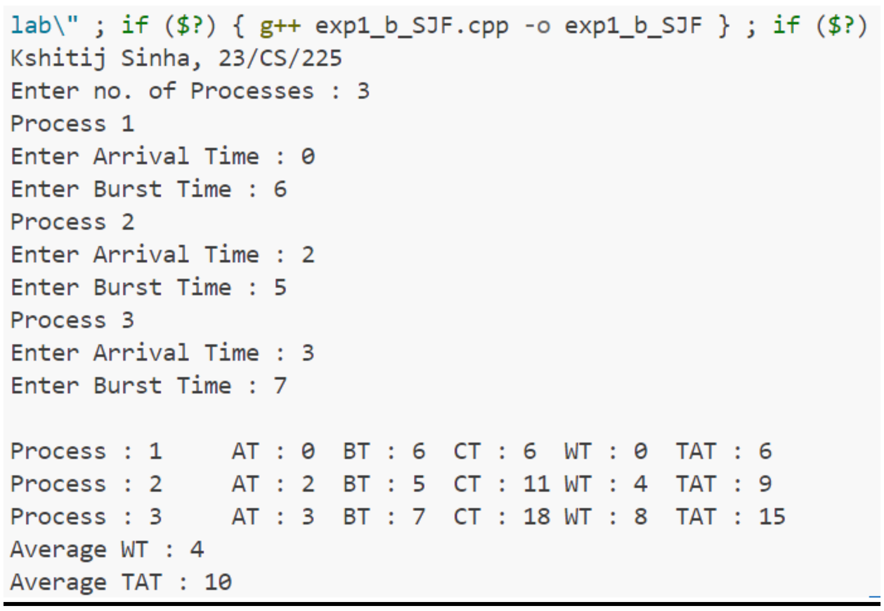
```
Kshitij SInha, 23/CS/225

Enter the no of processes in your system:
3

Enter arrival time for process 1:0
Enter burst time for process 1:4

Enter arrival time for process 2:5
Enter burst time for process 2:7

Enter arrival time for process 3:3
Enter burst time for process 3:9
Process Arrival Burst   Waiting Turn Around     Completion
 P[1]    0       4       0       4               4
 P[3]    3       9       1       10              13
 P[2]    5       7       8       15              20
```

Output of the FCFS program, containing AT, BT, WT, TAT, CT of processes

CODE(SJF)

```
#include <bits/stdc++.h>
using namespace std;

struct data1{
    int process;
    int AT, BT, CT, WT, TAT;
};

class SJF{
    int n;
    int time;
    double average_WT;
    double average_TAT;
    vector<data1> Data;

    struct comp{
        bool operator()(pair<data1, int> &a, pair<data1, int> &b){
            if (a.first.BT == b.first.BT)
                return a.first.AT > b.first.AT;
            return a.first.BT > b.first.BT;
        }
    };

    priority_queue<pair<data1, int>, vector<pair<data1, int>>, comp> pq;

    static bool cmp(data1 a, data1 b){
        return a.AT < b.AT;
    }

    static bool cmp1(data1 a, data1 b){
        return a.process < b.process;
    }

public:
    SJF(){
        time = 0;
        average_TAT = average_WT = 0;
    }

    void getnum(){
        cout << "Enter no. of Processes : ";
        cin >> n;
        Data.resize(n);
    }
    void getdata(){
        for (int i = 0; i < n; i++){
            Data[i].process = i + 1;
            cout << "Process " << Data[i].process << endl;
            cout << "Enter Arrival Time : ";
            cin >> Data[i].AT;
            cout << "Enter Burst Time : ";
            cin >> Data[i].BT;
        }

        sort(Data.begin(), Data.end(), cmp);
    }

    void find_CT(){
        pq.push({Data[0], 0});
        int index = 1;
        while (!pq.empty()){
            auto p = pq.top();
            pq.pop();
            auto task = p.first;
            int i = p.second;
            time = max(time, task.AT);
            time += task.BT;
            task.CT = time;
            Data[i] = task;

            while (index < n && Data[index].AT <= time){
                pq.push({Data[index], index});
                index++;
            }

            if (pq.empty() && index < n){
                pq.push({Data[index], index});
                index++;
            }
        }
    }

    void find_WT_TAT(){
        find_CT();

        for (int i = 0; i < n; i++){
            Data[i].TAT = Data[i].CT - Data[i].AT;
            Data[i].WT = Data[i].TAT - Data[i].BT;
            average_TAT += Data[i].TAT;
            average_WT += Data[i].WT;
        }

        average_TAT /= (double)n;
        average_WT /= (double)n;
    }

    void display(){
        sort(Data.begin(), Data.end(), cmp1);
        cout << endl;
        for (int i = 0; i < n; i++){
            cout << "Process : " << Data[i].process << '\t';
            cout << "AT : " << Data[i].AT << '\t';
            cout << "BT : " << Data[i].BT << '\t';
            cout << "CT : " << Data[i].CT << '\t';
            cout << "WT : " << Data[i].WT << '\t';
            cout << "TAT : " << Data[i].TAT << '\t';
            cout << endl;
        }
        cout << "Average WT : " << average_WT << endl;
        cout << "Average TAT : " << average_TAT << endl;
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    SJF sjf1;
    sjf1.getnum();
    sjf1.getdata();
    sjf1.find_WT_TAT();
    sjf1.display();
    return 0;
}
```

OUTPUT:



```
lab\" ; if ($?) { g++ exp1_b_SJF.cpp -o exp1_b_SJF } ; if ($?)
Kshitij Sinha, 23/CS/225
Enter no. of Processes : 3
Process 1
Enter Arrival Time : 0
Enter Burst Time : 6
Process 2
Enter Arrival Time : 2
Enter Burst Time : 5
Process 3
Enter Arrival Time : 3
Enter Burst Time : 7

Process : 1     AT : 0  BT : 6  CT : 6  WT : 0  TAT : 6
Process : 2     AT : 2  BT : 5  CT : 11 WT : 4  TAT : 9
Process : 3     AT : 3  BT : 7  CT : 18 WT : 8  TAT : 15
Average WT : 4
Average TAT : 10
```

Output of the SJF program, containing AT, BT, WT, TAT, CT of processes

OBSERVATIONS/RESULTS

FCFS has the possibility of convoy effect in which some processes which arrive earlier have long burst times, leading to starvation. It can lead to poor average waiting time. On, the other hand SJF is optimal for short jobs, but it requires a-priori information of the burst times of the processes which may not be available.

CONCLUSION

In conclusion, both non-preemptive FCFS and SJF have distinct strengths and limitations. FCFS is straightforward and fair by treating all processes equally in the order of arrival, making it suitable for simpler systems or workloads without varying burst times. However, it can lead to inefficiencies due to the convoy effect, which increases the average waiting time, especially when there is a mix of long and short processes.

Experiment 2

AIM

Write a C program to simulate the following pre-emptive CPU scheduling algorithms to find turnaround time and waiting time for a given problem. a) Round Robin b) Priority

THEORY

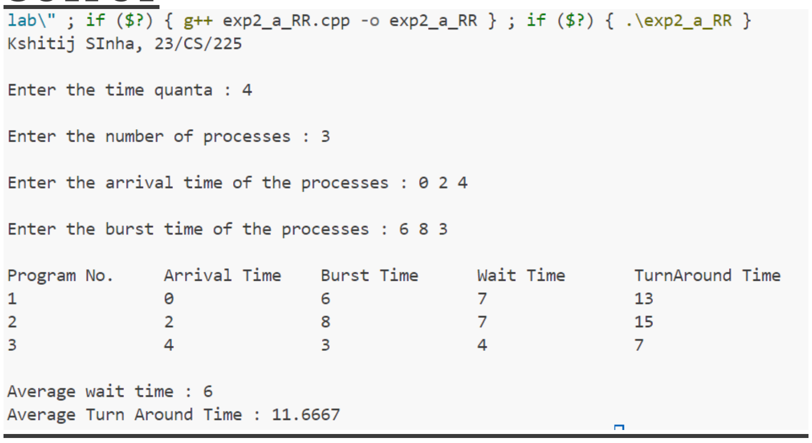
Pre-emptive CPU scheduling algorithms do terminate or put the process in a waiting queue after checking for some condition at regular time intervals. Round Robin assigns each process a fixed time slice, enabling fair CPU sharing, while Priority scheduling allocates CPU based on process priority. We calculate turnaround and waiting times for each.

CODE(RR)

```
#include <bits/stdc++.h>
using namespace std;
void queueUpdation(int queue[],int timer,int arrival[],int n, int maxProccessIndex){
    int zeroIndex;
    for(int i = 0; i < n; i++){
        if(queue[i] == 0){
            zeroIndex = i;
            break;
        }
    }
    queue[zeroIndex] = maxProccessIndex + 1;
}

void queueMaintainence(int queue[], int n){
    for(int i = 0; (i < n-1) && (queue[i+1] != 0) ; i++){
        int temp = queue[i];
        queue[i] = queue[i+1];
        queue[i+1] = temp;
    }
}

void checkNewArrival(int timer, int arrival[], int n, int maxProccessIndex,int queue[]){
    if(timer <= arrival[n-1]){
    bool newArrival = false;
    for(int j = (maxProccessIndex+1); j < n; j++){
            if(arrival[j] <= timer){
            if(maxProccessIndex < j){
                maxProccessIndex = j;
                newArrival = true;
            }
        }
    }
    if(newArrival)
        queueUpdation(queue,timer,arrival,n, maxProccessIndex);
    }
}
```

```
int main(){
    cout<<"Kshitij SInha, 23/CS/225"<<endl;
    int n,tq, timer = 0, maxProccessIndex = 0;
    float avgWait = 0, avgTT = 0;
    cout << "\nEnter the time quanta : ";
    cin>>tq;
    cout << "\nEnter the number of processes : ";
    cin>>n;
    int arrival[n], burst[n], wait[n], turn[n], queue[n], temp_burst[n];
    bool complete[n];

    cout << "\nEnter the arrival time of the processes : ";
    for(int i = 0; i < n; i++)
        cin>>arrival[i];

    cout << "\nEnter the burst time of the processes : ";
    for(int i = 0; i < n; i++){
        cin>>burst[i];
        temp_burst[i] = burst[i];
    }

    for(int i = 0; i < n; i++){
        complete[i] = false;
        queue[i] = 0;
    }
    while(timer < arrival[0])
        timer++;
    queue[0] = 1;

    while(true){
        bool flag = true;
        for(int i = 0; i < n; i++){
            if(temp_burst[i] != 0){
                flag = false;
                break;
            }
        }
        if(flag)
            break;

        for(int i = 0; (i < n) && (queue[i] != 0); i++){
            int ctr = 0;
            while((ctr < tq) && (temp_burst[queue[0]-1] > 0)){
                temp_burst[queue[0]-1] -= 1;
                timer += 1;
                ctr++;
                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
            }
            if((temp_burst[queue[0]-1] == 0) && (complete[queue[0]-1] == false)){
                turn[queue[0]-1] = timer;
                complete[queue[0]-1] = true;
            }

            bool idle = true;
            if(queue[n-1] == 0){
                for(int i = 0; i < n && queue[i] != 0; i++){
                    if(complete[queue[i]-1] == false){
                        idle = false;
                    }
                }
            }
            else
                idle = false;

            if(idle){
                timer++;
                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);
            }

            queueMaintainence(queue,n);
        }
    }

    for(int i = 0; i < n; i++){
        turn[i] = turn[i] - arrival[i];
        wait[i] = turn[i] - burst[i];
    }

    cout << "\nProgram No.\tArrival Time\tBurst Time\tWait Time\tTurnAround Time"
        << endl;
    for(int i = 0; i < n; i++){
        cout<<i+1<<"\t\t"<<arrival[i]<<"\t\t"
        <<burst[i]<<"\t\t"<<wait[i]<<"\t\t"<<turn[i]<<endl;
    }
    for(int i =0; i< n; i++){
        avgWait += wait[i];
        avgTT += turn[i];
    }
    cout<<"\nAverage wait time : "<<(avgWait/n)
    <<"\nAverage Turn Around Time : "<<(avgTT/n);
    return 0;
}
```

OUTPUT



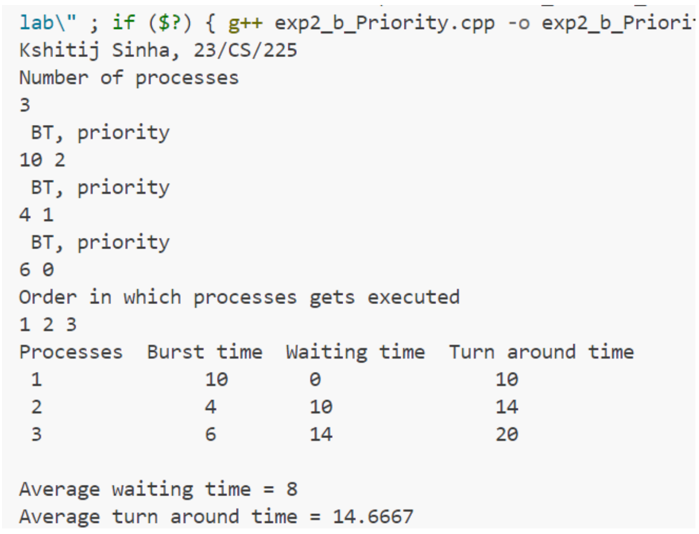
```
lab\" ; if ($?) { g++ exp2_a_RR.cpp -o exp2_a_RR } ; if ($?) { .\exp2_a_RR }
Kshitij SInha, 23/CS/225

Enter the time quanta : 4

Enter the number of processes : 3

Enter the arrival time of the processes : 0 2 4

Enter the burst time of the processes : 6 8 3

Program No.     Arrival Time    Burst Time      Wait Time       TurnAround Time
1               0               6               7               13
2               2               8               7               15
3               4               3               4               7

Average wait time : 6
Average Turn Around Time : 11.6667
```

Output of the RR program, containing AT, BT, WT, TAT, CT of processes

CODE(PRIORITY)

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
    int pid;
    int bt;
    int priority;
};

bool comparison(Process a, Process b) {
    return (a.priority > b.priority);
}

void findWaitingTime(Process proc[], int n, int wt[]) {
    wt[0] = 0;

    for (int i = 1; i < n; i++)
        wt[i] = proc[i - 1].bt + wt[i - 1];
}

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 << "\nProcesses "
        << " 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 " << 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;
}

void priorityScheduling(Process proc[], int n) {
    sort(proc, proc + n, comparison);
    cout << "Order in which processes gets executed \n";
    for (int i = 0; i < n; i++)
        cout << proc[i].pid << " ";
    findavgTime(proc, n);
}

int main() {
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int n;
    cout<<"Number of processes"<<endl;
    cin>>n;
    Process proc[n];
    for(int i=0; i<n; i++){
        proc[i].pid=i+1;
        cout<<" BT, priority"<<endl;
        cin>>proc[i].bt>>proc[i].priority;
    }
    priorityScheduling(proc, n);
    return 0;
}
```

OUTPUT:



```
lab\" ; if ($?) { g++ exp2_b_Priority.cpp -o exp2_b_Priori
Kshitij Sinha, 23/CS/225
Number of processes
3
 BT, priority
10 2
 BT, priority
4 1
 BT, priority
6 0
Order in which processes gets executed
1 2 3
Processes  Burst time  Waiting time  Turn around time
 1              10       0               10
 2              4        10              14
 3              6        14              20

Average waiting time = 8
Average turn around time = 14.6667
```

Output of the Priority program, containing BT, WT, TAT, CT of processes

OBSERVATIONS/RESULTS

Both these scheduling algorithms are significant improvements upon the non-pre-emptive algorithms as they lead to a higher throughput and efficiency. If the time quantum does not follow the 80% rule in Round Robin, the system will incur heavy overhead as a result of high-frequency swapping. Similarly for Priority scheduling, starvation may occur and a process may never get the resources it needs.

CONCLUSION

Pre-emptive Round Robin (RR) ensures fairness by allocating fixed time slices to each process, suitable for interactive tasks. Priority scheduling focuses on higher-priority tasks, offering efficient handling of critical processes but risking starvation for low-priority ones.

Experiment 3

AIM

Write a C program to simulate the following contiguous memory allocation techniques a) Worst fit, b) Best fit, c) First fit

THEORY

In this experiment, we implement a C program to simulate three contiguous memory allocation techniques:

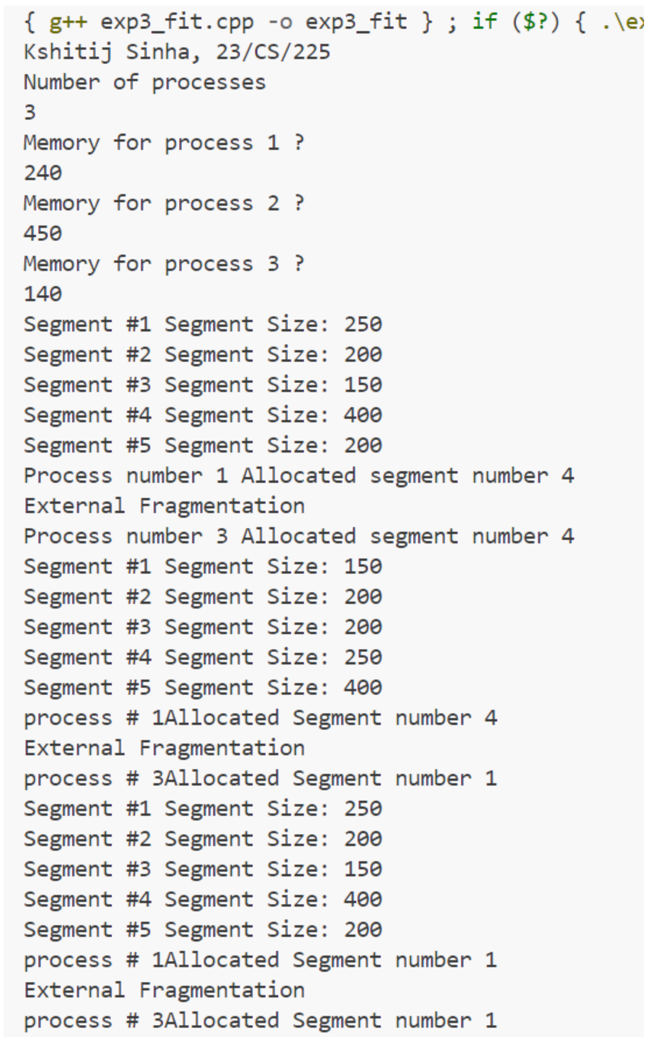
1. **Worst Fit**: Allocates the largest available memory block to a process, aiming to leave substantial free space for future allocations.
2. **Best Fit**: Allocates the smallest block that can fit the process, minimizing wasted space but often leading to smaller fragmented blocks.
3. **First Fit**: Allocates the first block that is large enough for the process, providing faster allocation but without prioritizing block size.

We construct a program with three functions that do these said procedures, and print out the metrics. Using basic sorting, searching, maintenance etc.

CODE

```
#include <bits/stdc++.h>
using namespace std;
void worst_allocate(vector<pair<int, int>> inp){
    //return the order in which processes get executed
    int mem[]={250, 200, 150, 400, 200};
    for(int i=0; i<5; i++){
        cout<<"Segment #"<<i+1<<" Segment Size: "<<mem[i]<<endl;
    }
    for(int i=0; i<inp.size(); i++){
        if(inp[i].second>400){
            cout<<"External Fragmentation"<<endl;
        }
        else{
            cout<<"Process number "<<i+1<<" Allocated segment number 4"<<endl;
        }
    }
}

void best_allocate(vector<pair<int, int>> inp){
    int mem[]={150, 200, 200, 250, 400};
    for(int i=0; i<5; i++){
        cout<<"Segment #"<<i+1<<" Segment Size: "<<mem[i]<<endl;
    }
    for(int i=0; i<inp.size(); i++){
        if(inp[i].second>400){
            cout<<"External Fragmentation"<<endl;
        }
        else{
            for(int j=0; j<5; j++){
                if(mem[j]>=inp[i].second){
                    cout<<"process # "<<i+1<<"Allocated Segment number "<<j+1<<endl;
                    break;
                }
                }
            }
        }
    }

void first_allocated(vector<pair<int, int>> inp){
    int mem[]={250, 200, 150, 400, 200};
    for(int i=0; i<5; i++){
        cout<<"Segment #"<<i+1<<" Segment Size: "<<mem[i]<<endl;
    }
    for(int i=0; i<inp.size(); i++){
        if(inp[i].second>400){
            cout<<"External Fragmentation"<<endl;
        }
        else{
            for(int j=0; j<5; j++){
                if(mem[j]>=inp[i].second){
                    cout<<"process # "<<i+1<<"Allocated Segment number "<<j+1<<endl;
                    break;
                }
            }
        }
    }
}
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int n;
    cout<<"Number of processes "<<endl;
    cin>>n;
    vector<pair<int, int>> inp;
    for(int i=0; i<n; i++){
        int mem;
        cout<<"Memory for process "<<i+1<<" ?"<<endl;
        cin>>mem;
        pair<int, int> temp;
        temp.first=i+1;
        temp.second=mem;
        inp.push_back(temp);
    }
    worst_allocate(inp);
    best_allocate(inp);
    first_allocated(inp);
}
```

OUTPUT



```
{ g++ exp3_fit.cpp -o exp3_fit } ; if ($?) { .\e
Kshitij Sinha, 23/CS/225
Number of processes
3
Memory for process 1 ?
240
Memory for process 2 ?
450
Memory for process 3 ?
140
Segment #1 Segment Size: 250
Segment #2 Segment Size: 200
Segment #3 Segment Size: 150
Segment #4 Segment Size: 400
Segment #5 Segment Size: 200
Process number 1 Allocated segment number 4
External Fragmentation
Process number 3 Allocated segment number 4
Segment #1 Segment Size: 150
Segment #2 Segment Size: 200
Segment #3 Segment Size: 200
Segment #4 Segment Size: 250
Segment #5 Segment Size: 400
process # 1Allocated Segment number 4
External Fragmentation
process # 3Allocated Segment number 1
Segment #1 Segment Size: 250
Segment #2 Segment Size: 200
Segment #3 Segment Size: 150
Segment #4 Segment Size: 400
Segment #5 Segment Size: 200
process # 1Allocated Segment number 1
External Fragmentation
process # 3Allocated Segment number 1
```

The segments allocated according to the three policies, assuming 5 holes in the memory

OBSERVATION/RESULTS

The code implements three memory allocation strategies: Worst Fit, Best Fit, and First Fit. Each function displays segment sizes and allocates memory to processes based on their requirements. In Worst Fit, if a process exceeds the largest segment, it reports external fragmentation. Best Fit allocates the smallest suitable segment, while First Fit allocates the first available segment.

It is a static implementation which assumes that the processes immediately release the memory after they are allocated.

CONCLUSION

In conclusion, the code effectively demonstrates three memory allocation techniques—Worst Fit, Best Fit, and First Fit.

- **Worst Fit** can minimize fragmentation but may leave larger gaps, making it inefficient for small processes.
- **Best Fit** optimizes space usage but can lead to small, unusable fragments.
- **First Fit** offers fast allocation but may create uneven fragmentation.

Enhancements like memory management and deallocation are needed to improve practical applicability and accuracy.

Experiment 4

AIM

Write a C program to simulate the following file allocation strategies a) Sequential, b) Indexed

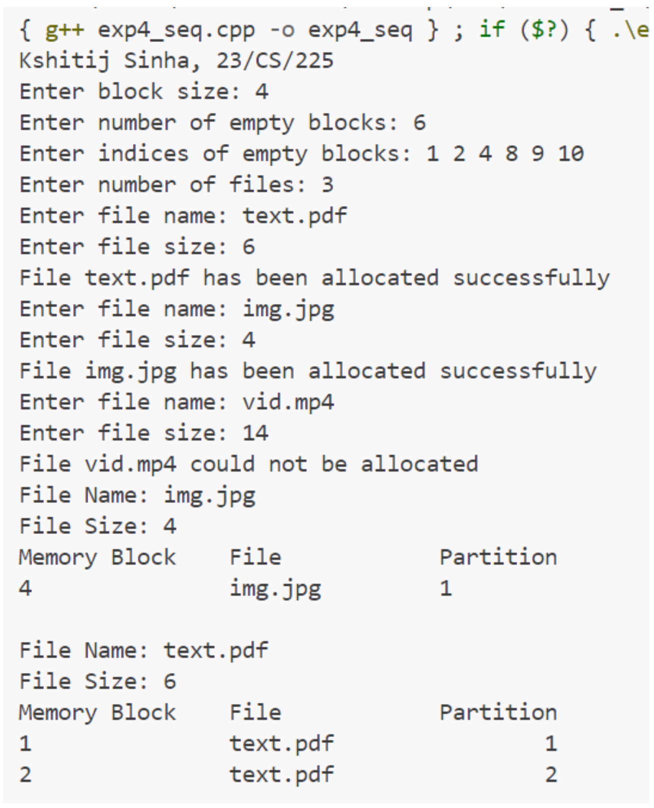
THEORY

Sequential allocation stores files in contiguous blocks for efficient access, while Indexed allocation uses an index block to manage file locations, allowing for non-contiguous storage and faster retrieval of data. Sequential allocation is easier to implement but can lead to inefficient allocation, while Indexed allocation can lead to more efficient allocation.

CODE(SEQUENTIAL)

```
# include <bits/stdc++.h>
using namespace std;

int block_size;
const int disk_size = 20;
int empty_blocks = 0;
int num_files = 0;
map<string, pair<int, int>> directory;
vector<bool> disk_avail(20, false);
vector<pair<string, int>> disk(20);

class file{
    public:
    string file_name;
    int file_size;
    int partitions;

    file() {};
    file(string &fname, int fsize) : file_name(fname), file_size(fsize) {
        partitions = (file_size + block_size - 1)/block_size;
    };
};

void initialise(){
    cout<<"Enter block size: "; cin>>block_size;
    cout<<"Enter number of empty blocks: "; cin>>empty_blocks;
    cout<<"Enter indices of empty blocks: ";
    for(int i = 0; i < empty_blocks; i++){
        int x; cin>>x;
        disk_avail[x] = true;
    }
}

void sequential_file_allocation(){
    cout<<"Enter number of files: "; cin>>num_files;
    for(int i = 0; i < num_files; i++){
        string fname; int fsize;
        cout<<"Enter file name: "; cin>>fname;
        cout<<"Enter file size: "; cin>>fsize;
        file f(fname, fsize);
        bool avail = false; int count = 0, startblock = -1;
        for(int j = 0; j < 20; j++){
            if(disk_avail[j]) count++;
            else count = 0;
            if(count == 1) startblock = j;
            if(count == f.partitions){
                avail = true; break;
            }
        }
        if(avail){
            directory[f.file_name] = {startblock, f.file_size};
            int count = 1;
            for(int j = startblock; count <= f.partitions; j++, count++){
                disk_avail[j] = false;
                disk[j] = {f.file_name, count};
            }
            cout<<"File "<<f.file_name<<" has been allocated successfully"<<endl;
        }
        else{
            cout<<"File "<<f.file_name<<" could not be allocated"<<endl;
        }
    }
}

void display_directory(){
    for(pair<string, pair<int, int>> entry: directory){
        string fname = entry.first; int startblock = entry.second.first;
        int fsize = entry.second.second;
        int endblock = startblock + (fsize + block_size - 1)/block_size - 1;
        cout<<"File Name: "<<fname<<endl;
        cout<<"File Size: "<<fsize<<endl;
        cout<<"Memory Block\tFile\t\tPartition"<<endl;
        for(int j = startblock; j <= endblock; j++){
            cout<<j<<"\t\t"<<disk[j].first<<"\t\t"<<disk[j].second<<endl;
        }
        cout<<endl;
    }
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    initialise();
    sequential_file_allocation();
    display_directory();
}
```

OUTPUT



```
{ g++ exp4_seq.cpp -o exp4_seq } ; if ($?) { .\e
Kshitij Sinha, 23/CS/225
Enter block size: 4
Enter number of empty blocks: 6
Enter indices of empty blocks: 1 2 4 8 9 10
Enter number of files: 3
Enter file name: text.pdf
Enter file size: 6
File text.pdf has been allocated successfully
Enter file name: img.jpg
Enter file size: 4
File img.jpg has been allocated successfully
Enter file name: vid.mp4
Enter file size: 14
File vid.mp4 could not be allocated
File Name: img.jpg
File Size: 4
Memory Block    File            Partition
4               img.jpg         1

File Name: text.pdf
File Size: 6
Memory Block    File            Partition
1               text.pdf                1
2               text.pdf                2
```

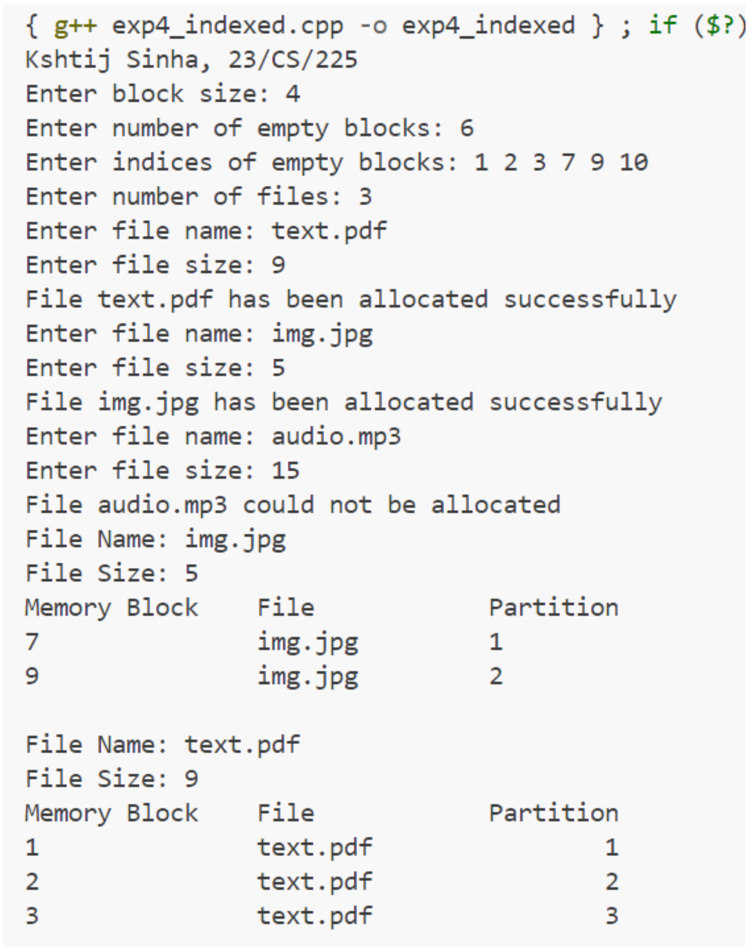
Allocation of files of various sizes and names, displaying the memory blocks which they occupy

CODE(INDEXED)

```
# include <bits/stdc++.h>
using namespace std;

int block_size;
const int disk_size = 20;
int empty_blocks = 0;
int num_files = 0;
map<string, pair<int, int>> directory;
vector<bool> disk_avail(disk_size, false);
vector<pair<string, int>> disk(disk_size);
vector<vector<int>> index_table;

class file{
    public:
    string file_name;
    int file_size;
    int partitions;

    file() {};
    file(string &fname, int fsize) : file_name(fname), file_size(fsize) {
        partitions = (file_size + block_size - 1)/block_size;
    };
};

void initialise(){
    cout<<"Enter block size: "; cin>>block_size;
    cout<<"Enter number of empty blocks: "; cin>>empty_blocks;
    cout<<"Enter indices of empty blocks: ";
    for(int i = 0; i < empty_blocks; i++){
        int x; cin>>x;
        disk_avail[x] = true;
    }
}

void indexed_file_allocation(){
    cout<<"Enter number of files: "; cin>>num_files;
    for(int i = 0; i < num_files; i++){
        string fname; int fsize;
        cout<<"Enter file name: "; cin>>fname;
        cout<<"Enter file size: "; cin>>fsize;
        file f(fname, fsize);
        int avail = 0;
        for(int j = 0; j < disk_size; j++){
            if(disk_avail[j]) avail++;
            if(avail >= f.partitions) break;
        }
        if(avail >= f.partitions){
            directory[f.file_name] = {index_table.size(), f.file_size};
            vector<int> index_table_entry; int count = 0;
            for(int j = 0; count < f.partitions; j++){
                if(disk_avail[j]){
                    disk_avail[j] = false;
                    disk[j] = {f.file_name, count + 1};
                    index_table_entry.push_back(j);
                    count++;
                }
            }
            index_table.push_back(index_table_entry);
            cout<<"File "<<f.file_name<<" has been allocated successfully"<<endl;
        }
        else{
            cout<<"File "<<f.file_name<<" could not be allocated"<<endl;
        }
    }
}

void display_directory(){
    for(pair<string, pair<int, int>> entry: directory){
        string fname = entry.first; int file_index = entry.second.first;
        int fsize = entry.second.second;
        cout<<"File Name: "<<fname<<endl;
        cout<<"File Size: "<<fsize<<endl;
        cout<<"Memory Block\tFile\t\tPartition"<<endl;
        int fpartitions = index_table[file_index].size();
        for(int j = 0; j < fpartitions; j++){
            int block = index_table[file_index][j];
            cout<<block<<"\t\t"<<disk[block].first<<"\t\t"<<disk[block].second<<endl;
        }
        cout<<endl;
    }
}
```

```cpp
int main(){
    cout<<"Kshtij Sinha, 23/CS/225"<<endl;
    initialise();
    indexed_file_allocation();
    display_directory();
}
```

OUTPUT



```
{ g++ exp4_indexed.cpp -o exp4_indexed } ; if ($?)
Kshtij Sinha, 23/CS/225
Enter block size: 4
Enter number of empty blocks: 6
Enter indices of empty blocks: 1 2 3 7 9 10
Enter number of files: 3
Enter file name: text.pdf
Enter file size: 9
File text.pdf has been allocated successfully
Enter file name: img.jpg
Enter file size: 5
File img.jpg has been allocated successfully
Enter file name: audio.mp3
Enter file size: 15
File audio.mp3 could not be allocated
File Name: img.jpg
File Size: 5
Memory Block    File            Partition
7               img.jpg         1
9               img.jpg         2

File Name: text.pdf
File Size: 9
Memory Block    File            Partition
1               text.pdf                1
2               text.pdf                2
3               text.pdf                3
```

Allocation of files of various sizes and names, displaying the memory blocks which they occupy

OBSERVATIONS/RESULTS

The code implements two file allocation strategies: **Sequential** and **Indexed**.

1. **Sequential Allocation** efficiently allocates contiguous blocks, minimizing access time but potentially leading to fragmentation. It tracks allocated blocks using bit mapping of the segment blocks.
2. **Indexed Allocation** supports non-contiguous storage, improving space utilization with an index table for each file. It also tracks allocated blocks using bit mapping to keep track of the segments hat are free and those which have been allocated.

Both implementations demonstrate fundamental concepts in file allocation with distinct advantages and challenges.

CONCLUSION

In conclusion, the code effectively simulates two file allocation strategies: Sequential and Indexed.

Sequential allocation offers fast access with contiguous storage but risks fragmentation, while Indexed allocation allows for more flexible, non-contiguous storage, improving space utilization. Both methods illustrate key concepts in file management, highlighting their respective strengths and weaknesses, and suggesting potential areas for improvement in handling edge cases and optimizing performance.

Experiment 5

AIM

Write a C program to simulate Banker’s algorithm for the purpose of Deadlock avoidance.

THEORY

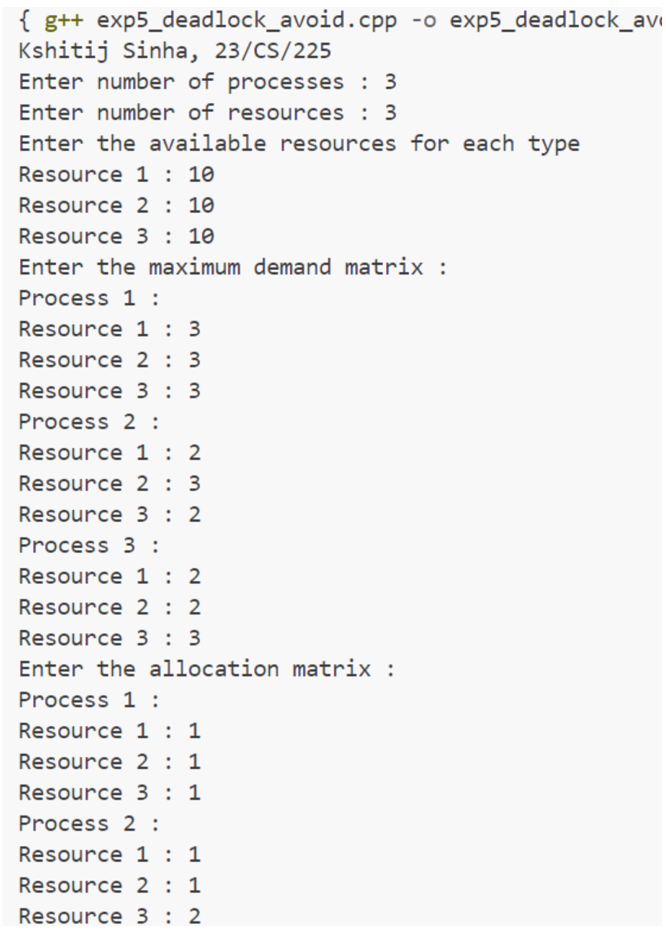
The Banker's algorithm is a deadlock avoidance strategy used in operating systems to allocate resources safely. It simulates resource allocation based on current requests and maximum needs, ensuring the system remains in a safe state. This prevents deadlocks by evaluating if requests can be fulfilled without compromising resource availability.

CODE

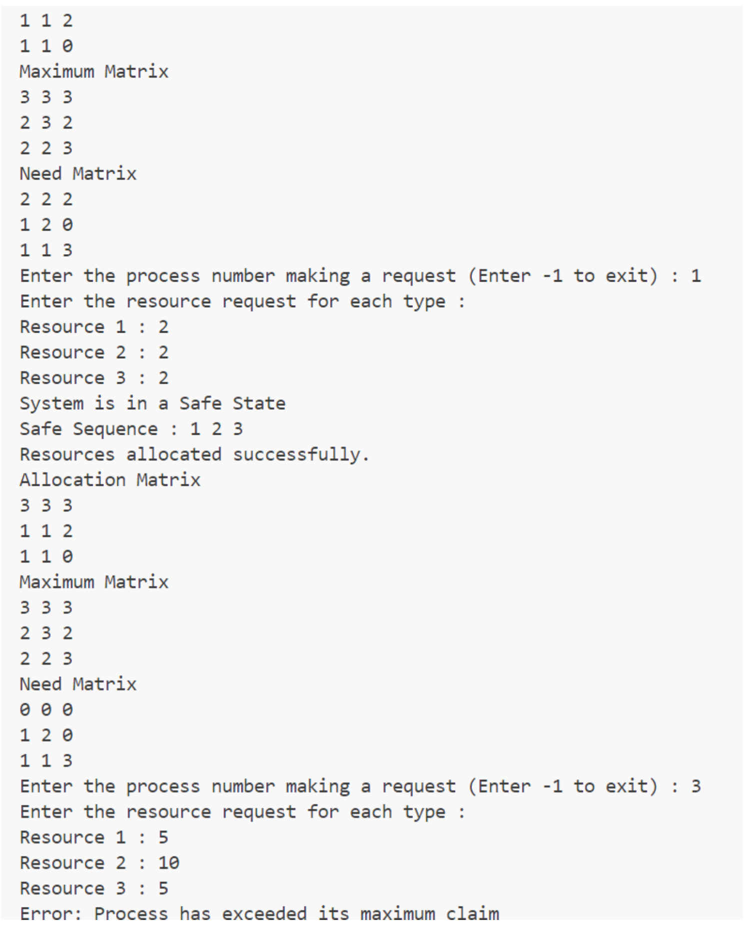
```
#include <bits/stdc++.h>
using namespace std;

class Bankers_Algorithm_Avoidance
{
    vector<int> available;
    vector<vector<int>> maximum;
    vector<vector<int>> allocation;
    vector<vector<int>> need;

    int processes, resources;

    void calculate_need()
    {
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                need[i][j] = maximum[i][j] - allocation[i][j];
            }
        }
    }

public:
    void display_allocation()
    {
        cout << "Allocation Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << allocation[i][j] << " ";
            }
            cout << endl;
        }
    }

    void display_maximum()
    {
        cout << "Maximum Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << maximum[i][j] << " ";
            }
            cout << endl;
        }
    }

    void display_need()
    {
        cout << "Need Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << need[i][j] << " ";
            }
            cout << endl;
        }
    }

    void getnum()
    {
        cout << "Enter number of processes : ";
        cin >> processes;
        cout << "Enter numnber of resources : ";
        cin >> resources;

        available.resize(resources);
        maximum.resize(processes, vector<int>(resources));
        allocation.resize(processes, vector<int>(resources));
        need.resize(processes, vector<int>(resources));
    }

    void getdata()
    {
        cout << "Enter the available resources for each type" << endl;
        for (int i = 0; i < resources; i++)
        {
            cout << "Resource " << i + 1 << " : ";
            cin >> available[i];
        }

        cout << "Enter the maximum demand matrix : " << endl;
        for (int i = 0; i < processes; i++)
        {
            cout << "Process " << i + 1 << " : " << endl;
            for (int j = 0; j < resources; j++)
            {
                cout << "Resource " << j + 1 << " : ";
                cin >> maximum[i][j];
            }
        }
        cout << "Enter the allocation matrix : " << endl;
        for (int i = 0; i < processes; i++)
        {
            cout << "Process " << i + 1 << " : " << endl;
            for (int j = 0; j < resources; j++)
            {
                cout << "Resource " << j + 1 << " : ";
                cin >> allocation[i][j];
            }
        }

        calculate_need();

        display_allocation();
        display_maximum();
        display_need();
    }

    // Check if a process can be safely allocated resources
    bool can_allocate(int process, vector<int> &work)
    {
        for (int i = 0; i < resources; i++)
        {
            if (need[process][i] > work[i])
            {
                return false;
            }
        }
        return true;
    }

    // Function to check if the system is in a safe state
    bool isSafe()
    {
        vector<int> work = available;
        vector<bool> finish(processes, false);
        vector<int> safe_sequence;

        for (int count = 0; count < processes; count++)
        {
            for (int i = 0; i < processes; i++)
            {
                if (!finish[i] && can_allocate(i, work))
                {
                    for (int j = 0; j < resources; j++)
                    {
                        work[j] += allocation[i][j];
                    }
                    safe_sequence.push_back(i + 1);
                    finish[i] = true;
                }
            }
        }
```

```
        bool found = true;
        for (int i = 0; i < processes; i++)
        {
            if (!finish[i])
            {
                found = false;
                break;
            }
        }

        if (!found)
        {
            cout << "System is not in a Safe State" << endl;
            return false;
        }

        cout << "System is in a Safe State" << endl;
        cout << "Safe Sequence : ";
        for (int i = 0; i < safe_sequence.size(); i++)
        {
            cout << safe_sequence[i] << " ";
        }
        cout << endl;
        return true;
    }

    // Request resources for a process
    bool request_resources(int process, vector<int> request)
    {
        for (int i = 0; i < resources; i++)
        {
            if (request[i] > need[process][i])
            {
                cout << "Error: Process has exceeded its maximum claim" << endl;
                return false;
            }
            if (request[i] > available[i])
            {
                cout << "Resources not available" << endl;
                return false;
            }
        }

        // Tentatively allocate the resources
        for (int i = 0; i < resources; i++)
        {
            available[i] -= request[i];
            allocation[process][i] += request[i];
            need[process][i] -= request[i];
        }

        // Check if the system remains in a safe state
        if (isSafe())
        {
            cout << "Resources allocated successfully." << endl;

            display_allocation();
            display_maximum();
            display_need();

            return true;
        }
        else
        {
            // Rollback allocation
            for (int i = 0; i < resources; i++)
            {
                available[i] += request[i];
                allocation[process][i] -= request[i];
                need[process][i] += request[i];
            }
            cout << "Resources Allocation leads to an Unsafe State." << endl;
            return false;
        }
    }

    void take_request()
    {
        int process;
        vector<int> request(resources);
        while (true)
        {
            cout << "Enter the process number making a request (Enter -1 to exit) : ";
            cin >> process;

            if (process == -1)
                break;

            process--;
            cout << "Enter the resource request for each type : " << endl;
            for (int i = 0; i < resources; i++)
            {
                cout << "Resource " << i + 1 << " : ";
                cin >> request[i];
            }

            request_resources(process, request);
        }
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    Bankers_Algorithm_Avoidance ba;
    ba.getnum();
    ba.getdata();
    ba.take_request();
    return 0;
}
```

OUTPUT



```
{ g++ exp5_deadlock_avoid.cpp -o exp5_deadlock_av
Kshitij Sinha, 23/CS/225
Enter number of processes : 3
Enter number of resources : 3
Enter the available resources for each type
Resource 1 : 10
Resource 2 : 10
Resource 3 : 10
Enter the maximum demand matrix :
Process 1 :
Resource 1 : 3
Resource 2 : 3
Resource 3 : 3
Process 2 :
Resource 1 : 2
Resource 2 : 3
Resource 3 : 2
Process 3 :
Resource 1 : 2
Resource 2 : 2
Resource 3 : 3
Enter the allocation matrix :
Process 1 :
Resource 1 : 1
Resource 2 : 1
Resource 3 : 1
Process 2 :
Resource 1 : 1
Resource 2 : 1
Resource 3 : 2
```



```
1 1 2
1 1 0
Maximum Matrix
3 3 3
2 3 2
2 2 3
Need Matrix
2 2 2
1 2 0
1 1 3
Enter the process number making a request (Enter -1 to exit) : 1
Enter the resource request for each type :
Resource 1 : 2
Resource 2 : 2
Resource 3 : 2
System is in a Safe State
Safe Sequence : 1 2 3
Resources allocated successfully.
Allocation Matrix
3 3 3
1 1 2
1 1 0
Maximum Matrix
3 3 3
2 3 2
2 2 3
Need Matrix
0 0 0
1 2 0
1 1 3
Enter the process number making a request (Enter -1 to exit) : 3
Enter the resource request for each type :
Resource 1 : 5
Resource 2 : 10
Resource 3 : 5
Error: Process has exceeded its maximum claim
```

Output demonstrating the allocated, maximum, and need matrices; then taking user commands to decided whether allocation is safe/possible

OBSERVATIONS/RESULTS

The code implements the Banker's algorithm for deadlock avoidance, effectively managing resource allocation among processes. It calculates the need matrix dynamically and checks system safety through after each resource request. The modular structure and user prompts enhance usability, making it a clear demonstration of the Banker's algorithm principles in resource management and deadlock prevention.

CONCLUSION

In conclusion, the Banker's algorithm is a critical deadlock avoidance strategy in operating systems, ensuring safe resource allocation among processes. By assessing the maximum resource needs and current availability, it maintains system stability. This approach effectively prevents deadlocks by allowing resource requests only when they can be satisfied without jeopardizing the system's safe state, thereby facilitating efficient multitasking and resource management.

Experiment 6

AIM

Write a C program to simulate Banker's algorithm for the purpose of Deadlock prevention.

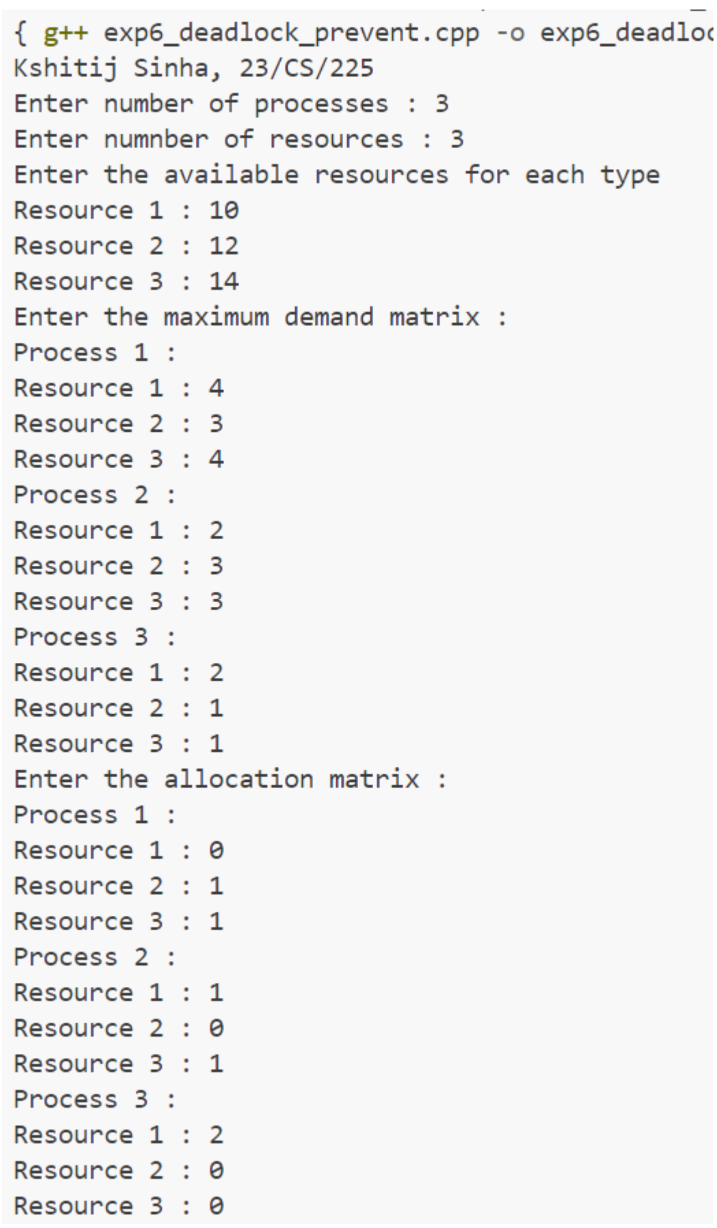
THEORY

A deadlock can be prevented by making sure that any one of the four conditions for deadlock formation are not met (Mutual Exclusion, Hold and Wait, No Pre-emption, Circular Wait) It evaluates resource requests by comparing the maximum demand of processes with current availability. By ensuring that resources are allocated only if the system remains in a safe state, it effectively prevents deadlocks.

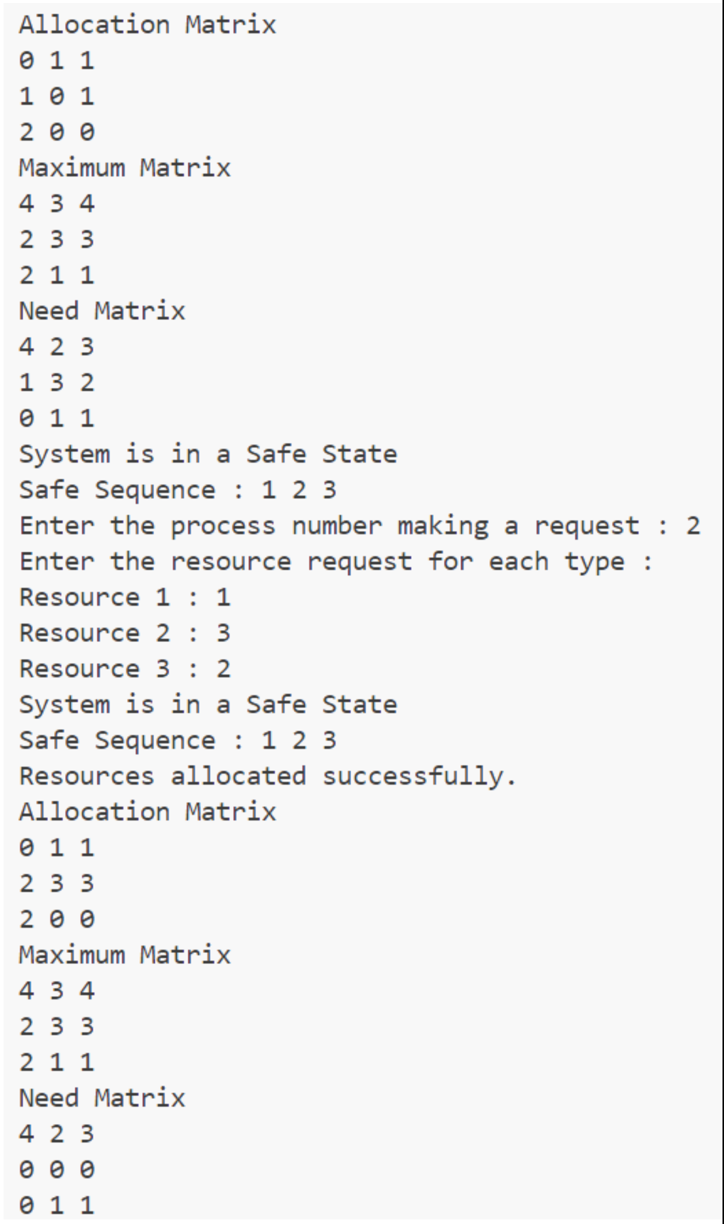
CODE

```
#include <bits/stdc++.h>
using namespace std;

class Bankers_Algorithm_Prevention
{
    vector<int> available;
    vector<vector<int>> maximum;
    vector<vector<int>> allocation;
    vector<vector<int>> need;

    int processes, resources;

    void calculate_need()
    {
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                need[i][j] = maximum[i][j] - allocation[i][j];
            }
        }
    }

public:
    void display_allocation()
    {
        cout << "Allocation Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << allocation[i][j] << " ";
            }
            cout << endl;
        }
    }

    void display_maximum()
    {
        cout << "Maximum Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << maximum[i][j] << " ";
            }
            cout << endl;
        }
    }

    void display_need()
    {
        cout << "Need Matrix" << endl;
        for (int i = 0; i < processes; i++)
        {
            for (int j = 0; j < resources; j++)
            {
                cout << need[i][j] << " ";
            }
            cout << endl;
        }
    }

    void getnum()
    {
        cout << "Enter number of processes : ";
        cin >> processes;
        cout << "Enter numnber of resources : ";
        cin >> resources;

        available.resize(resources);
        maximum.resize(processes, vector<int>(resources));
        allocation.resize(processes, vector<int>(resources));
        need.resize(processes, vector<int>(resources));
    }

    void getdata()
    {
        cout << "Enter the available resources for each type" << endl;
        for (int i = 0; i < resources; i++)
        {
            cout << "Resource " << i + 1 << " : ";
            cin >> available[i];
        }

        cout << "Enter the maximum demand matrix : " << endl;
        for (int i = 0; i < processes; i++)
        {
            cout << "Process " << i + 1 << " : " << endl;
            for (int j = 0; j < resources; j++)
            {
                cout << "Resource " << j + 1 << " : ";
                cin >> maximum[i][j];
            }
        }
        cout << "Enter the allocation matrix : " << endl;
        for (int i = 0; i < processes; i++)
        {
            cout << "Process " << i + 1 << " : " << endl;
            for (int j = 0; j < resources; j++)
            {
                cout << "Resource " << j + 1 << " : ";
                cin >> allocation[i][j];
            }
        }

        calculate_need();

        display_allocation();
        display_maximum();
        display_need();
    }

    // Check if a process can be safely allocated resources
    bool can_allocate(int process, vector<int> &work)
    {
        for (int i = 0; i < resources; i++)
        {
            if (need[process][i] > work[i])
            {
                return false;
            }
        }
        return true;
    }

    // Function to check if the system is in a safe state
    bool isSafe()
    {
        vector<int> work = available;
        vector<bool> finish(processes, false);
        vector<int> safe_sequence;

        for (int count = 0; count < processes; count++)
        {
            for (int i = 0; i < processes; i++)
            {
                if (!finish[i] && can_allocate(i, work))
                {
                    for (int j = 0; j < resources; j++)
                    {
                        work[j] += allocation[i][j];
                    }
                    safe_sequence.push_back(i + 1);
                    finish[i] = true;
                }
            }
        }
        bool found = true;
        for (int i = 0; i < processes; i++)
        {
            if (!finish[i])
            {
                found = false;
                break;
            }
        }

        if (!found)
        {
            cout << "System is not in a Safe State" << endl;
            return false;
        }

        cout << "System is in a Safe State" << endl;
        cout << "Safe Sequence : ";
        for (int i = 0; i < safe_sequence.size(); i++)
        {
            cout << safe_sequence[i] << " ";
        }
        cout << endl;
        return true;
    }

    // Request resources for a process
    void request_resources(int process, vector<int> request)
    {
        for (int i = 0; i < resources; i++)
        {
            if (request[i] > need[process][i])
            {
                cout << "Error: Process has exceeded its maximum claim" << endl;
                return;
            }
            if (request[i] > available[i])
            {
                cout << "Resources not available" << endl;
                return;
            }
        }

        // Tentatively allocate the resources
        for (int i = 0; i < resources; i++)
        {
            available[i] -= request[i];
            allocation[process][i] += request[i];
            need[process][i] -= request[i];
        }

        // Check if the system remains in a safe state
        if (isSafe())
        {
            cout << "Resources allocated successfully." << endl;

            display_allocation();
            display_maximum();
            display_need();
        }
        else
        {
            // Rollback allocation
            for (int i = 0; i < resources; i++)
            {
                available[i] += request[i];
                allocation[process][i] -= request[i];
                need[process][i] += request[i];
            }
            cout << "Resources Allocation leads to an Unsafe State." << endl;
        }
    }

    void take_request()
    {
        int process;
        vector<int> request(resources);
        cout << "Enter the process number making a request : ";
        cin >> process;
        process--;
        cout << "Enter the resource request for each type : " << endl;
        for (int i = 0; i < resources; i++)
        {
            cout << "Resource " << i + 1 << " : ";
            cin >> request[i];
        }

        request_resources(process, request);
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    Bankers_Algorithm_Prevention ba;
    ba.getnum();
    ba.getdata();

    if (ba.isSafe())
    {
        ba.take_request();
    }
    return 0;
}
```

OUTPUT



```
{ g++ exp6_deadlock_prevent.cpp -o exp6_deadloc
Kshitij Sinha, 23/CS/225
Enter number of processes : 3
Enter numnber of resources : 3
Enter the available resources for each type
Resource 1 : 10
Resource 2 : 12
Resource 3 : 14
Enter the maximum demand matrix :
Process 1 :
Resource 1 : 4
Resource 2 : 3
Resource 3 : 4
Process 2 :
Resource 1 : 2
Resource 2 : 3
Resource 3 : 3
Process 3 :
Resource 1 : 2
Resource 2 : 1
Resource 3 : 1
Enter the allocation matrix :
Process 1 :
Resource 1 : 0
Resource 2 : 1
Resource 3 : 1
Process 2 :
Resource 1 : 1
Resource 2 : 0
Resource 3 : 1
Process 3 :
Resource 1 : 2
Resource 2 : 0
Resource 3 : 0
```



```
Allocation Matrix
0 1 1
1 0 1
2 0 0
Maximum Matrix
4 3 4
2 3 3
2 1 1
Need Matrix
4 2 3
1 3 2
0 1 1
System is in a Safe State
Safe Sequence : 1 2 3
Enter the process number making a request : 2
Enter the resource request for each type :
Resource 1 : 1
Resource 2 : 3
Resource 3 : 2
System is in a Safe State
Safe Sequence : 1 2 3
Resources allocated successfully.
Allocation Matrix
0 1 1
2 3 3
2 0 0
Maximum Matrix
4 3 4
2 3 3
2 1 1
Need Matrix
4 2 3
0 0 0
0 1 1
```

Decides whether the allocation is safe, and the safe sequence if it's safe, upon analysing user inputs of processes and their requirements

OBSERVATIONS/RESULTS

The Banker's algorithm focuses on maintaining a safe state during resource allocation to prevent deadlocks. It dynamically calculates the need matrix based on maximum demands and current allocations, allowing efficient resource management. By incorporating user requests, it enhances understanding and supports multitasking while ensuring system stability. We have basically incorporated the pseudocode algorithm in the book into a working program with an interface.

CONCLUSION

In conclusion, the Banker's algorithm is a vital tool for deadlock prevention in operating systems, ensuring safe resource allocation among processes. By dynamically assessing resource requests against maximum demands, it prevents unsafe states and deadlocks.

Experiment 7

AIM:

Write a C program to simulate page replacement algorithm a) FIFO, b) LRU

THEORY

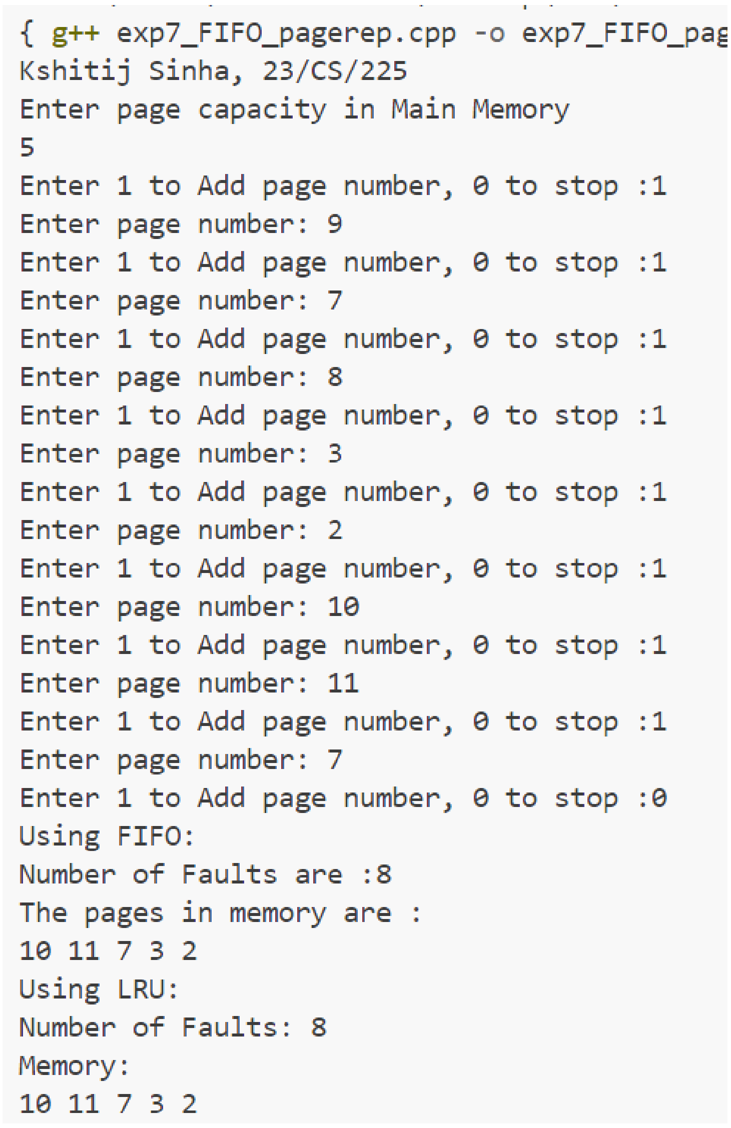
Non-contiguous allocation of memory to the processes is implemented though the use of pages and frames. The physical memory is divided into fixed sized frames and the logical memory is divided into pages. Pages are mapped to frames. When there is no more memory left, a page-fault occurs and some page must be removed to make space for the incoming page. A page-fault also occurs when the space is unoccupied and a new page needs to occupy the space. Page replacement algorithms are used to resolve this:

- **FIFO (First-In, First-Out)**: FIFO is a page replacement strategy where the oldest page in memory is replaced first during a page fault. It uses a queue to manage pages, adding new pages at the rear and removing the oldest from the front.
- **LRU (Least Recently Used)**: LRU replaces the page that hasn't been used for the longest time. It tracks page usage to evict the least recently accessed page during a fault.

CODE

```
//Input: page numbers, number of pages in memory
//Output: number of page faults, final pages in memory
#include <bits/stdc++.h>
using namespace std;
class fifo{
    public:
    fifo(int n, vector<int> p){
        int faults=0;
        int mem[n];
        for(int i=0; i<n; i++){
            mem[i]=-1;
        }
        int prev=0;
        for(int i=0; i<p.size(); i++){
            if(i<n){
                 bool found=false;
                for(int j=0; j<n; j++){
                    if(mem[j]==p[i]){
                        found=true;
                        break;
                    }
                }
                if(found==false){
                mem[i]=p[i];
                faults++;
                }
            }
            else{
                bool found=false;
                for(int j=0; j<n; j++){
                    if(mem[j]==p[i]){
                        found=true;
                        break;
                    }
                }
                if(found==false){
                    for(int j=0; j<n; j++){
                        if(mem[j]==p[prev]){
                            mem[j]=p[i];
                            prev++;
                            faults++;
                            break;
                        }
                    }
                }
            }
        }
        cout<<"Number of Faults are :"<<faults<<endl;
        cout<<"The pages in memory are :"<<endl;
        for(int i=0; i<n; i++){
            cout<<mem[i]<<" ";
        }
    }
};

class lru{
    public:
    lru(int n, vector<int> p){
        int mem[n];
        map<int, int> mpp;
        for(int i=0; i<p.size(); i++){
            mpp[p[i]]++;
        }
        vector<int> accessed;
        for(int i=0; i<n; i++){
            mem[i]=-1;
        }
        int faults=0;
        for(int i=0; i<p.size(); i++){
            if(i<n){
                bool avail=false;
                for(int j=0; j<n; j++){
                    if(mem[j]==p[i]){
                        avail=true;
                        accessed.push_back(accessed[0]);
                        accessed.erase(accessed.begin());
                        break;
                    }
                }
                if(avail==true){
                    continue;
                }
                else{
                    accessed.push_back(p[i]);
                    mem[i]=p[i];
                    faults++;
                }
            }
            else{
                int search=accessed[0];
                for(int j=0; j<n; j++){
                    if(mem[j]==search){
                        mem[j]=p[i];
                        faults++;
                    }
                }
                accessed.erase(accessed.begin());
                accessed.push_back(p[i]);
            }
        }
        cout<<"Number of Faults: "<<faults<<endl;
        cout<<"Memory: "<<endl;
        for(int i=0; i<n; i++){
            cout<<mem[i]<<" ";
        }
        cout<<endl;
    }
};


int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int n;
    cout<<"Enter page capacity in Main Memory"<<endl;
    cin>>n;
    vector<int> pages;
    while(1){
        cout<<"Enter 1 to Add page number, 0 to stop :";
        int x;
        cin>>x;
        if(x==1){
            cout<<"Enter page number: ";
            int p;
            cin>>p;
            pages.push_back(p);
        }
        else{
            break;
        }
    }
    cout<<"Using FIFO: "<<endl;
    fifo f1(n, pages);
    cout<<endl;
    cout<<"Using LRU: "<<endl;
    lru l1(n, pages);
}
```

OUTPUT



```
{ g++ exp7_FIFO_pagerep.cpp -o exp7_FIFO_pag
Kshitij Sinha, 23/CS/225
Enter page capacity in Main Memory
5
Enter 1 to Add page number, 0 to stop :1
Enter page number: 9
Enter 1 to Add page number, 0 to stop :1
Enter page number: 7
Enter 1 to Add page number, 0 to stop :1
Enter page number: 8
Enter 1 to Add page number, 0 to stop :1
Enter page number: 3
Enter 1 to Add page number, 0 to stop :1
Enter page number: 2
Enter 1 to Add page number, 0 to stop :1
Enter page number: 10
Enter 1 to Add page number, 0 to stop :1
Enter page number: 11
Enter 1 to Add page number, 0 to stop :1
Enter page number: 7
Enter 1 to Add page number, 0 to stop :0
Using FIFO:
Number of Faults are :8
The pages in memory are :
10 11 7 3 2
Using LRU:
Number of Faults: 8
Memory:
10 11 7 3 2
```

Demonstrates both FIFO and LRU, giving the image of memory and the number of page faults as well

OBSERVATIONS/RESULTS

The program has used queues to maintain the lru and fifo properties, it handles the exceptions of a page already present in the page table. It also uses a global variable for the number of page faults. FIFO is a straightforward algorithm, but it may lead to Belady's anomaly, leading to more page faults with increased memory frames.

LRU keeps track of the order of page accesses and replaces the least recently used page upon a fault. This method often results in fewer page faults compared to FIFO because it retains pages that are frequently accessed.

CONCLUSION

Both algorithms provide valuable insights into memory management strategies in operating systems, with LRU generally outperforming FIFO in scenarios with frequent reuse of certain pages. The results of page faults can vary significantly based on the page reference string and the size of memory.

Experiment 8

AIM

Write a C program to simulate page replacement algorithm a) LFU, b) Optimal

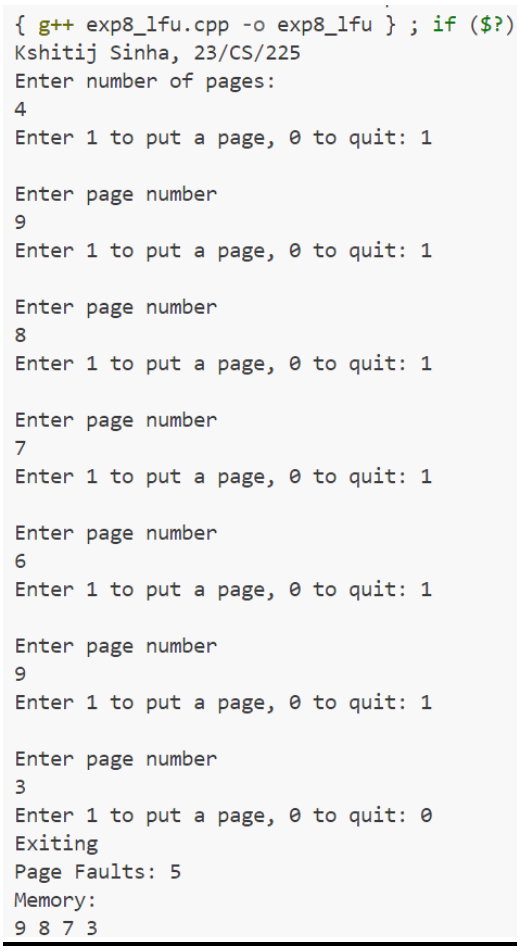
THEORY

- **LFU (Least Frequently Used) Page Replacement Algorithm:** LFU replaces the page with the least access frequency. It tracks how often each page is used, and upon a page fault, it removes the page with the lowest count.
- **Optimal Page Replacement Algorithm:** The Optimal algorithm replaces the page that will not be needed for the longest future duration. Although it requires knowledge of future requests, making it impractical, it serves as an ideal benchmark, ensuring the fewest page faults compared to other algorithms.

CODE (LFU)

```
//Input: page numbers, number of pages in memory
//Output: number of page faults, final pages in memory
//Least frquent first
#include <bits/stdc++.h>
using namespace std;
class lfu{
    public:
    map<int, int>unique, accessed;
    lfu(int n, vector<int> p){
        int mem[n];
        for(int i=0; i<n; i++){
            mem[i]=-1;
        }
        for(int i=0; i<p.size(); i++){
            unique[p[i]]++;
        }
        for(int i=0; i<p.size(); i++){
            if(i<n){
                bool found=false;
                for(int j=0; j<n; j++){
                    if(mem[j]==p[i]){
                        found=true;
                        break;
                    }
                }
                if(found==true){
                    continue;
                }
                mem[i]=p[i];
                accessed[p[i]]++;
            }
            else{
                int recent;
                for(auto it: accessed){
                    recent=it.first;
                    break;
                }
                for(int j=0; j<n; j++){
                    if(mem[j]==recent){
                        mem[j]=p[i];
                        accessed[p[i]]++;
                    }
                }
            }
        }
        cout<<"Page Faults: "<<unique.size()<<endl;
        cout<<"Memory: "<<endl;
        for(int i=0; i<n; i++){
            cout<<mem[i]<<" ";
        }
        cout<<endl;
    }
};
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int n;
    cout<<"Enter number of pages: "<<endl;
    cin>>n;
    vector<int>p;
    while(1){
        cout<<"Enter 1 to put a page, 0 to quit: ";
        int x;
        cin>>x;
        if(x==1){
            cout<<endl;
            cout<<"Enter page number"<<endl;
            int y;
            cin>>y;
            p.push_back(y);
        }
        else{
            cout<<"Exiting"<<endl;
            break;
        }
    }
    lfu l(n, p);
}
```

OUTPUT

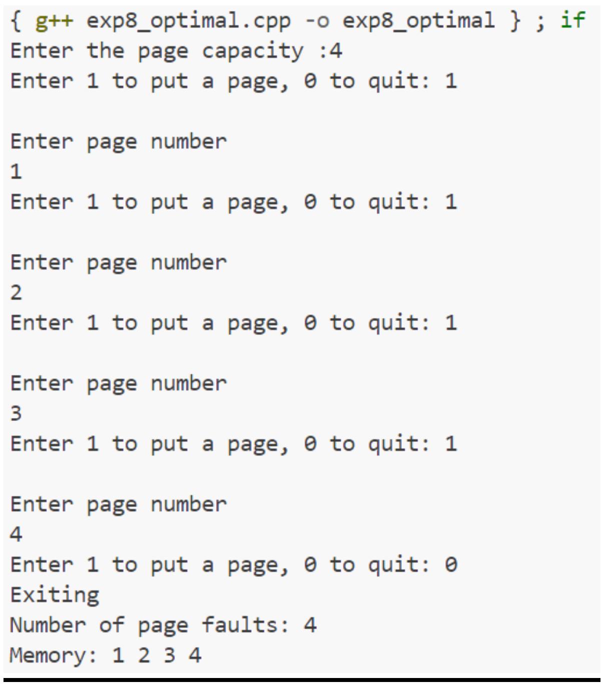


```
{ g++ exp8_lfu.cpp -o exp8_lfu } ; if ($?)
Kshitij Sinha, 23/CS/225
Enter number of pages:
4
Enter 1 to put a page, 0 to quit: 1

Enter page number
9
Enter 1 to put a page, 0 to quit: 1

Enter page number
8
Enter 1 to put a page, 0 to quit: 1

Enter page number
7
Enter 1 to put a page, 0 to quit: 1

Enter page number
6
Enter 1 to put a page, 0 to quit: 1

Enter page number
9
Enter 1 to put a page, 0 to quit: 1

Enter page number
3
Enter 1 to put a page, 0 to quit: 0
Exiting
Page Faults: 5
Memory:
9 8 7 3
```

CODE(OPTIMAL)

```cpp
#include <bits/stdc++.h>
using namespace std;
bool search(int x, int mem[], int n){
    for(int i=0; i<n; i++){
        if(mem[i]==x){
            return true;
        }
    }
    return false;
}
class optimal{
    public:

    optimal(vector<int>p, int n){
        int mem[n];
        for(int i=0; i<n; i++){
            mem[i]=-1;
        }
        for(int i=0; i<p.size(); i++){
            if(search(p[i], mem, n)){
                continue;
            }
            if(i<n){
                mem[i]=p[i];
            }
            else{
                int forward[n];
                for(int t=0; t<n; t++){
                    forward[i]=-1;
                }
                int maxx=INT_MIN;
                for(int j=0; j<n; j++){
                    for(int k=i+1; k<p.size(); k++){
                        if(p[k]==mem[j]){
                            forward[j]=k;
                            maxx=max(k, maxx);
                        }
                    }
                }
                bool found=false;
                for(int j=0; j<n; j++){
                    if(forward[j]==-1){
                        mem[j]=p[i];
                        found=true;
                    }
                }
                if(found==true){
                    continue;
                }
                else{
                    for(int j=0; j<n; j++){
                        if(forward[j]==maxx){
                            mem[j]=p[i];
                            break;
                        }
                    }
                }
            }
        }
        map<int, int> mpp;
        for(int i=0; i<p.size(); i++){
            mpp[p[i]]++;
        }
        cout<<"Number of page faults: "<<mpp.size()<<endl;
        cout<<"Memory: ";
        for(int i=0; i<n; i++){
            cout<<mem[i]<<" ";
        }
        cout<<endl;
    }
};

int main(){
    int n;
    cout<<"Enter the page capacity :";
    cin>>n;
    vector<int>p;
    while(1){
        cout<<"Enter 1 to put a page, 0 to quit: ";
        int x;
        cin>>x;
        if(x==1){
            cout<<endl;
            cout<<"Enter page number"<<endl;
            int y;
            cin>>y;
            p.push_back(y);
        }
        else{
            cout<<"Exiting"<<endl;
            break;
        }
    }
    optimal(p, n);
}
```

OUTPUT



```
{ g++ exp8_optimal.cpp -o exp8_optimal } ; if
Enter the page capacity :4
Enter 1 to put a page, 0 to quit: 1

Enter page number
1
Enter 1 to put a page, 0 to quit: 1

Enter page number
2
Enter 1 to put a page, 0 to quit: 1

Enter page number
3
Enter 1 to put a page, 0 to quit: 1

Enter page number
4
Enter 1 to put a page, 0 to quit: 0
Exiting
Number of page faults: 4
Memory: 1 2 3 4
```

OBSERVATIONS/RESULTS

LFU was implemented through maintaining a map of frequency of all elements, if there are multiple elements of least frequency, LRU was applied on those elements to find a victim page. Optimal is a perfect algorithm which minimizes the number of page faults but isn’t practical as it requires a-priori information of all the pages and their sequencing.

CONCLUSION

LFU and Optimal algorithms offer unique advantages: LFU adapts to frequent access patterns, while Optimal minimizes page faults by anticipating future needs. Though impractical, Optimal serves as a benchmark for measuring other page replacement strategies.

Experiment 9

AIM

Write a C program to simulate producer-consumer problem using semaphores.

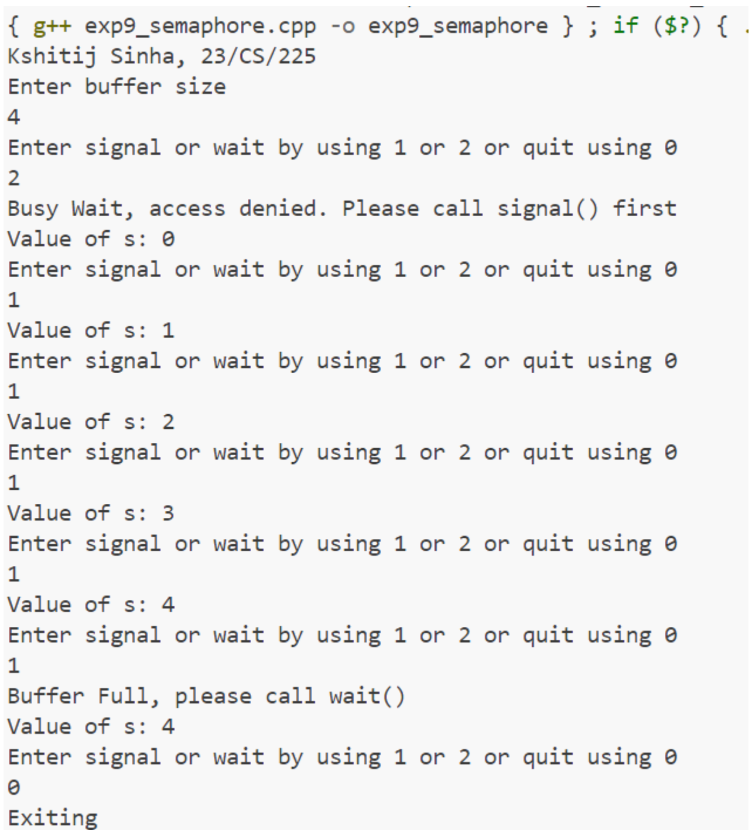
THEORY

In the producer-consumer problem with semaphores, two types of processes—producers adding items to a buffer and consumers removing items—need synchronized access to avoid race conditions. Two commands- wait() and signal() manage the allocation and deallocation of a common resource to a number of processes. A process enters busy wait if the value of the semaphore is less than 0

CODE

```
#include <bits/stdc++.h>
using namespace std;
void wait(int& s){
    if(s<=0){
        cout<<"Busy Wait, access denied. Please call signal() first"<<endl;
        cout<<"Value of s: "<<s<<endl;
        return;
    }
    s--;
    cout<<"Value of s: "<<s<<endl;
}

void signal(int& s, int& n){
    s++;
    if(s>n){
        cout<<"Buffer Full, please call wait()"<<endl;
        s--;
        cout<<"Value of s: "<<s<<endl;
        return;
    }
    cout<<"Value of s: "<<s<<endl;
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int n;
    cout<<"Enter buffer size"<<endl;
    cin>>n;
    int val=true;
    int s=0;
    while(1){
        cout<<"Enter signal or wait by using 1 or 2 or quit using 0"<<endl;
        int x;
        cin>>x;
        if(x==1){
            signal(s, n);
        }
        else if(x==2){
            wait(s);
        }
        else{
            cout<<"Exiting"<<endl;
            break;
        }
    }
}
```

OUTPUT



```
{ g++ exp9_semaphore.cpp -o exp9_semaphore } ; if ($?) { .
Kshitij Sinha, 23/CS/225
Enter buffer size
4
Enter signal or wait by using 1 or 2 or quit using 0
2
Busy Wait, access denied. Please call signal() first
Value of s: 0
Enter signal or wait by using 1 or 2 or quit using 0
1
Value of s: 1
Enter signal or wait by using 1 or 2 or quit using 0
1
Value of s: 2
Enter signal or wait by using 1 or 2 or quit using 0
1
Value of s: 3
Enter signal or wait by using 1 or 2 or quit using 0
1
Value of s: 4
Enter signal or wait by using 1 or 2 or quit using 0
1
Buffer Full, please call wait()
Value of s: 4
Enter signal or wait by using 1 or 2 or quit using 0
0
Exiting
```

Demonstrates the use of semaphore using the buffer, wait, and signal commands. Handles exceptions of overflow and underflow

OBSERVATIONS/RESULTS

The program handles the producer consumer problem by using a buffer of fixed size, it uses wait and signal to put in/out elements. It handles exceptions as well.

CONCLUSION

This semaphore code effectively manages process synchronization by controlling access to shared resources. Using `wait` to block on zero or negative values and `signal` to increment within buffer limits, it ensures orderly access and prevents buffer overflow.

Experiment 10

AIM

Write a C program to simulate disk scheduling algorithms a) FCFS, b) SCAN

THEORY

Disk scheduling optimizes the order in which disk I/O requests are served to improve efficiency.

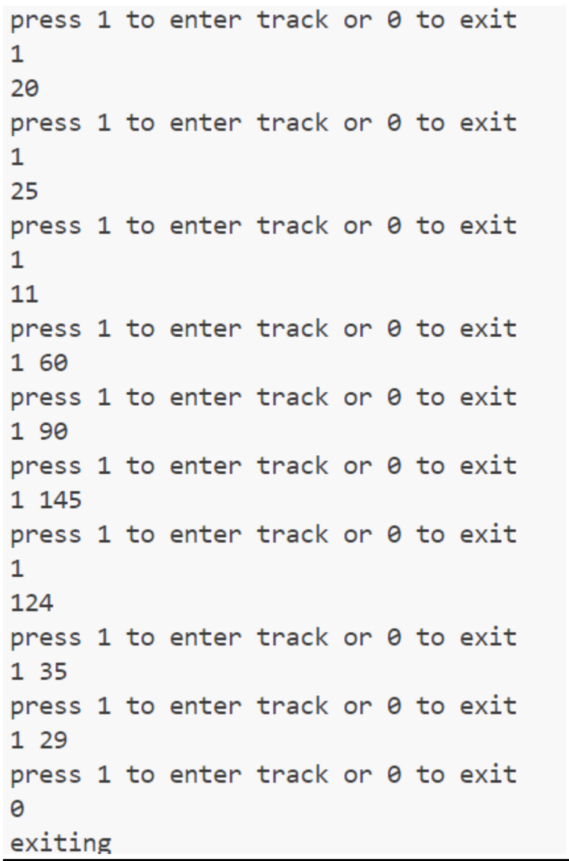
1. **FCFS (First-Come, First-Served):** Processes requests in the order they arrive, resulting in simplicity but potentially high seek time if requests are scattered across the disk.

2. **SCAN (Elevator Algorithm):** Moves the disk arm in one direction, serving all requests until reaching the disk's end, then reverses direction. This reduces seek time compared to FCFS, especially when requests are spread out.

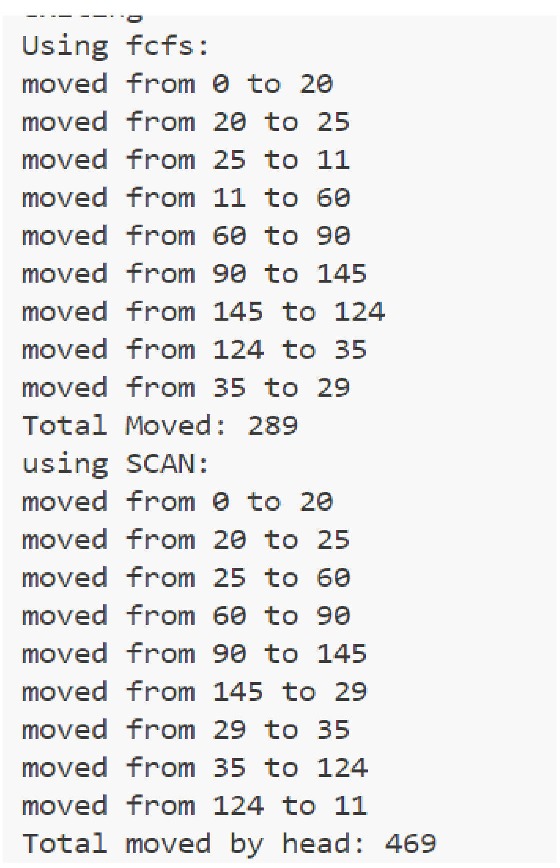
CODE

```
#include <bits/stdc++.h>
using namespace std;
class fcfs{
    public:
    fcfs(vector<int> p){
        int moved=0;
        moved+=abs(p[0]);
        cout<<"moved from "<<0<<" to "<<p[0]<<endl;
        for(int i=1; i<p.size(); i++){
            cout<<"moved from "<<p[i-1]<<" to "<<p[i]<<endl;
            moved+=abs(p[i]-p[i-1]);
        }
        cout<<"Total Moved: "<<moved<<endl;
    }
};

class scan{
    public:
    scan(vector<int> p){
        int moved=0;
        moved+=abs(p[0]);
        int head=p[0];
        int taken[p.size()]={0};
        cout<<"moved from "<<0<<" to "<<p[0]<<endl;
        for(int i=1; i<p.size(); i++){
            if(p[i]>=head){
                cout<<"moved from "<<head<<" to "<<p[i]<<endl;
                moved+=abs(p[i]-head);
                head=p[i];
                taken[i]=1;
            }
        }
        for(int i=p.size()-1; i>=1; i--){
            if(taken[i]==0){
                cout<<"moved from "<<head<<" to "<<p[i]<<endl;
                moved+=abs(p[i]-head);
                head=p[i];
            }
        }
        cout<<"Total moved by head: "<<moved<<endl;
    }
};

int main(){
    vector<int>p;
    while(1){
        cout<<"press 1 to enter track or 0 to exit"<<endl;
        int x;
        cin>>x;
        if(x==1){
            int y;
            cin>>y;
            p.push_back(y);
        }
        else{
            cout<<"exiting"<<endl;
            break;
        }
    }
    cout<<"Using fcfs: "<<endl;
    fcfs f(p);
    cout<<"using SCAN: "<<endl;
    scan s(p);
}
```

OUTPUT



```
press 1 to enter track or 0 to exit
1
20
press 1 to enter track or 0 to exit
1
25
press 1 to enter track or 0 to exit
1
11
press 1 to enter track or 0 to exit
1 60
press 1 to enter track or 0 to exit
1 90
press 1 to enter track or 0 to exit
1 145
press 1 to enter track or 0 to exit
1
124
press 1 to enter track or 0 to exit
1 35
press 1 to enter track or 0 to exit
1 29
press 1 to enter track or 0 to exit
0
exiting
```



```
Using fcfs:
moved from 0 to 20
moved from 20 to 25
moved from 25 to 11
moved from 11 to 60
moved from 60 to 90
moved from 90 to 145
moved from 145 to 124
moved from 124 to 35
moved from 35 to 29
Total Moved: 289
using SCAN:
moved from 0 to 20
moved from 20 to 25
moved from 25 to 60
moved from 60 to 90
moved from 90 to 145
moved from 145 to 29
moved from 29 to 35
moved from 35 to 124
moved from 124 to 11
Total moved by head: 469
```

Demonstrates the two disk scheduling algorithms, highlighting the movements and the total distance moved by the head

OBSERVATIONS/RESULTS

The distance moved by the heads depend on the sequencing of the addresses of the I/O commands in the disks. If adjacent entries are close by, FCFS is the better approach else the SCAN algorithm is the better, more efficient approach.

CONCLUSION

Both FCFS and SCAN algorithms have their pros and cons, their efficiencies depend on the sequencing of address of the I/O operations requested.