EXPERIMENT 1

Aim To implement FCFS scheduling technique

Theory

Operating System: A system software responsible for running a computer, managing all resources and implementing scheduling.

CPU scheduling: a process handled by the OS to ensure maximum CPU time utilization. There are 2 important queues maintained here, job queue and ready queue

FIRST COME FIRST SERVE (FCFS)

The simplest scheduling technique as it serves job to the processor in the order they are received. It is not efficient as waiting time and response time aren't considered

All scheduling techniques tend to efficiency between 40 to 90%

Also average waiting time and turn around time should less.

General Keywords:

Waiting Time : time taken till processor is allotted for the first time.

Response Time : time taken when processor gives some response out of process like some O/P or first response.

Turn around time : time taken when job is received and is terminated, i.e. time taken to complete its execution.

Burst Time : processor time required to complete a job.

Result
Successfully implemented FCFS scheduling technique.

EXPERIMENT 1 FCFS

```
#include<iostream>
using namespace std;

// waiting time function
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
	// waiting time for first process is 0
	wt[0] = 0;

	// calculating waiting time
	for (int i = 1; i < n ; i++ )
		wt[i] = bt[i-1] + wt[i-1] ;
}

// turn around time function
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[]) {
	// calculate turnaround time by adding
	// bt[i] + wt[i]
	for (int i = 0; i < n ; i++)
		tat[i] = bt[i] + wt[i];
}

// average time function
void findavgTime( int processes[], int n, int bt[]) {
	int wt[n], tat[n], total_wt = 0, total_tat = 0;

	findWaitingTime(processes, n, bt, wt);

	findTurnAroundTime(processes, n, bt, wt, tat);

	cout << "Processes "<< " Burst time "
		<< " Waiting time " << " Turn around time\n";

	// Calculate total waiting time and total turn
	// around time
	for (int i=0; i<n; i++) {
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		cout << " " << i+1 << "\t\t" << bt[i] <<"\t "<< wt[i] <<"\t\t " << tat[i] <<endl;
	}

	cout << "Average waiting time = "<< (float)total_wt / (float)n;
	cout << "\nAverage turn around time = "<< (float)total_tat / (float)n;
}


int main() {
	int processes[] = { 1, 2, 3};
	int n = sizeof processes / sizeof processes[0];
	int burst_time[] = {10, 5, 8};

	findavgTime(processes, n, burst_time);
	cout << "\n";
	return 0;
}
```

```
:~/Documents/COLLEGE SEM-6/OS/Practicals

:~/Documents/COLLEGE SEM-6/OS/Practicals
```

DELTA Pg No
Date / /

EXPERIMENT 2

Aim: Write a program to illustrate Shortest Job First Scheduling algorithm.

Theory Shortest Job First is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

Shortest Job First has the advantage of having a minimum average waiting time among all scheduling problems.

Algorithm

1) Sort all the process according to the arrival time.
2) Then select that process which has minimum arrival time and minimum Burst time.
3) After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which has minimum Burst time.

The pre emptive version of shortest job first is called the shortest remaining time first algorithm, in it the process with the smallest amount of time remaining completion is selected to execute.

EXPERIMENT 2 SJF

```
#include<iostream>
using namespace std;
int mat[10][6];

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void arrangeArrival(int num, int mat[][6]) {
    for(int i=0; i<num; i++) {
        for(int j=0; j<num-i-1; j++) {
            if(mat[j][1] > mat[j+1][1]) {
                for(int k=0; k<5; k++) {
                    swap(mat[j][k], mat[j+1][k]);
                }
            }
        }
    }
}

void completionTime(int num, int mat[][6]) {
    int temp, val;
    mat[0][3] = mat[0][1] + mat[0][2];
    mat[0][5] = mat[0][3] - mat[0][1];
    mat[0][4] = mat[0][5] - mat[0][2];

    for(int i=1; i<num; i++) {
        temp = mat[i-1][3];
        int low = mat[i][2];
        for(int j=i; j<num; j++) {
            if(temp >= mat[j][1] && low >= mat[j][2]) {
                low = mat[j][2];
                val = j;
            }
        }
        mat[val][3] = temp + mat[val][2];
        mat[val][5] = mat[val][3] - mat[val][1];
        mat[val][4] = mat[val][5] - mat[val][2];
        for(int k=0; k<6; k++) {
            swap(mat[val][k], mat[i][k]);
        }
    }
}

int main() {
    int num, temp;

    cout<<"Enter number of Process: ";
    cin>>num;
    cout<<" ... Enter the process ID ... \n";
    for(int i=0; i<num; i++) {
        cout<<" ... Process "<<i+1<<" ... \n";
        cout<<"Enter Process Id: ";
        cin>>mat[i][0];
        cout<<"Enter Arrival Time: ";
        cin>>mat[i][1];
```

```
        cout<<"Enter Burst Time: ";
        cin>>mat[i][2];
    }

    cout<<"Before Arrange...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\n";
    for(int i=0; i<num; i++) {
        cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\n";
    }

    arrangeArrival(num, mat);
    completionTime(num, mat);
    cout<<"Final Result...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\
n";
    for(int i=0; i<num; i++) {
        cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\t\t"<<mat[i]
[4]<<"\t\t"<<mat[i][5]<<"\n";
    }
      return 0;
}
```

```
kunal@Kunal-Notebook: [illegible]
~$ ./a.out
Enter number of Process: 4
...Enter the process ID...
...Process 1...
Enter Process Id: 1
Enter Arrival Time: 2
Enter Burst Time: 4
...Process 2...
Enter Process Id: 2
Enter Arrival Time: 1
Enter Burst Time: 4
...Process 3...
Enter Process Id:       3
Enter Arrival Time: 2
Enter Burst Time: 6
...Process 4...
Enter Process Id: 4
Enter Arrival Time: 3
Enter Burst Time: 2
Before Arrange...
Process ID      Arrival Time    Burst Time
1               2               4
2               1               4
3               2               6
4               3               2
Final Result...
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
2               1               4               0               4
4               3               2               2               4
1               2               4               5               9
3               2               6               9               15
```

DELTA Pg No
Date / /

EXPERIMENT 3

Aim: Write a program to perform priority scheduling.

Theory:

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on. Process with the same priority on FCFS basis.

The preemptive priority scheduling considers the arrival time of the processes. In this scheduling, the CPU can leave the process midway. The current state of the process will be saved by the context switch. The system can then search for another process with a higher priority in the ready queue or waiting queue and start its execution

Advantages

→ Priority scheduling is easy to implement
→ The aging technique is implemented to reduce the starvation of lower priority process.

Disadvantages:

→ Starvation of lower priority process.
→ Avg waiting time is higher in non pre-emptive

→ If a system failure occurs, all unfinished lower priority jobs get vanished from the system.

Result Successfully performed priority scheduling.

EXPERIMENT 3 PRIORITY SCHEDULING

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

struct Process{
      int pid;
      int priority;
      int burst;
      int arrival;
};

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

void waitTimes(Process process[], int n, int wt[]){
      wt[0]=0;
      for(int i=1;i<n;i++){
            wt[i]= process[i-1].burst + wt[i-1];
      }
}

void turnTimes(Process process[], int n, int wt[], int tt[]){
      tt[0]=process[0].burst;
      for(int i=1;i<n;i++){
            tt[i]= process[i].burst + wt[i];
      }
}

void findAvg(Process process[], int n){

      int wt[n], tt[n], totalWt=0, totalTt=0;
      waitTimes(process, n, wt);
      turnTimes(process, n, wt, tt);

      cout<<"\nProcesses\t"<<"Priority\t"<<"Burst\t\t"<<"Waiting\t\
t"<<"TurnAround";

      for(int i=0;i<n;i++){
            totalWt+=wt[i];
            totalTt+=tt[i];
            cout<<"\n"<<process[i].pid<<"\t\t"<<process[i].priority<<"\t\
t"<<process[i].burst<<"\t\t"<<wt[i]<<"\t\t"<<tt[i];
      }
      cout<<"\nAvg Waiting Times: "<<(float)totalWt/(float)n;
      cout<<"\nAvg Turn Around Times: "<<(float)totalTt/(float)n<<"\n";
}


void priority(Process proc[], int n){
      sort(proc, proc + n, comparison);
      findAvg(proc,n);
}


int main(){
      Process proc[] = {{1,1,2,1},{2,2,4,2},{3,3,3,5}};
      int n = sizeof(proc)/sizeof(proc[0]);
```

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

Output

```
                :~/Documents/COLLEGE SEM-6/OS/Practicals

                :~/Documents/COLLEGE SEM-6/OS/Practicals
```

EXPERIMENT 4

Aim: To implement Round Robin Scheduling.

Theory:

Round Robin Scheduling is used to schedule process fairly each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which are arrived in the quantum time that make these scheduling fairly.

Advantage:

→ Round robin is cyclic in nature so starvation doesn't occur
→ Round robin is variant of first come first serve
→ No priority, special importance given to any process or task.
→ RR scheduling is also known as Time Slicing Scheduling
→ Each process is served by CPU for a fixed time so priority is same

Dis Advantages:

→ Throughput depends on quantum time.
→ A process can't be prioritized over the other.

CODE

```
#include <iostream>
using namespace std;

int main() {
    cout << "Enter the number of processes: ";
    int nProc = 0; cin >> nProc;
    cout << "Enter burst times of processes: ";
    int data[10][5];
    for (int i = 0; i < nProc; ++i) {
        cin >> data[i][0];
        data[i][1] = data[i][0]; // burst time subtractions
        data[i][2] = 0;  // for waiting times
        data[i][3] = 0;  // for Turn around times
        data[i][4] = 0; // for status of processes: if complete
    }
    int time = 0;
    cout << "Enter time quantum: "; cin >> time;
    while (1) {
        bool resume = 0;
        for (int i=0; i<nProc; ++i) {
            if (data[i][4] == 0) {
                resume = 1; break;
            } if (resume) {
                for (int i=0; i< nProc; ++i) {
                    if (data[i][1] <= time) {
                        for (int j=0; j< nProc; ++j) {
```

```
            if (i==j) continue;
            else if (data[j][4] ==0)
            data[j][2] += data[i][1];
        }
        data [i][1]=0;
        data [i] [4]=1;
    }
    else if (data[i][1] > time){
        data [i][1] -= time;
        for(int j=0; j<nProc; ++j){
            if(i==j) continue;
            else if (data[j][4]==0)
            data[j][2] += time;
        }
    }
  }
} else if (!resume) break;
}
float avgWt=0, avgTat=0;
for (int i=0; i<nProc; ++i){
    data[i][3] = data[i][0] + data[i][2];
    avgWt += data [i][2];
    avgTat += data [i][3];
}
cout << "Average waiting time: " << avgWt << endl;
cout << "Average turn around time: " << avgTat << endl;
return 0;
}
```

OUTPUT

Enter the no.of processes: 3

Enter the burst times of processes : 2 4 3

Enter time quantum: 5

Process ID 0	Process ID 1	Process ID 2
Burst time: 2	Burst time : 4	Burst Time : 3
Waiting time : 0	Waiting time : 2	Waiting time : 6
TAT : 2	TAT : 6	TAT : 9

Average waiting time : 2.66667

Average turn around time : 5.66667

Result

Implemented Round Robin scheduling successfully.

EXPERIMENT 5

Aim Write a program for page replacement policy using a) LRU b) FIFO c) Optimal

Theory

In an algorithm that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page replacement Algorithms:

- First In First Out (FIFO) -
  The operating system keeps track of all pages in the memory in a queue. When a page is to be replaced, the page in front of the queue is selected for removal.

- Least Recently Used (LRU)
  The page which was least recently used will be replaced.

- Optimal Page Replacement
  The page which won't be used for the longest time is replaced.

DATE: / /
PAGE NO:

Program

a) Least Recently Used

```
#include <iostream>
#include< bits/stdc++.h>

using namespace std;

int pageFaults (int pages[], int n, int capacity)

    unordered_set<int> s;
    unordered_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;
                for (auto it = s.begin(); it != s.end(); it++){
                    if (indexes[*it ]< LRU) {
                        LRU = indexes[*it];
                        val = it;
                    }
                }
            }
```

Be Positive....

Sign.....

DATE: / /
PAGE NO:

```
            s.erase(val);
            s.insert(pages[i]);
            page_faults++;
        }
        indexes[pages[i]] = i;
    }
}
return page_faults;
}

int main() {
    int pages[] = {7,0,1,2,0,3,0,4,2,3,0,3,2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

OUTPUT:

6

b) FIRST IN FIRST OUT (FIFO)

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

int pageFaults(int pages[], int n, int capacity) {
    unordered_set<int> s;
    queue<int> indexes;

    int page_faults = 0;
    for(int i=0; i<n; i++){
```


Sign

DATE: / /
PAGE NO:

```
            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 pages[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,3};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

OUTPUT:

7

Sign

DATE: / /
PAGE NO:

(c) Optimals Page Replacement

```
#include <iostream>
#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;
}
```

Be Positive....

Sign.....

DATE / /
PAGE NO.

```
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 hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}
int main() {
    int pg[] = {7,0,1,2,0,3,0,4,2,3,0,3,2};
    int pn = sizeof(pg) / sizeof(pg[0]);
    int fn = 4;
    optimalPage(pg, pn, fn);
    return 0;
}
```

Output:

No. of hits = 7

No. of misses = 6

Result: Successfully implemented page replacement algorithms.

Sign

EXPERIMENT 6

AIM: Write a program to implement first fit, best fit and worst fit algorithm for memory management.

Theory

In an operating system, one of the memory management technique is partitioned allocation, in which memory is divided in different blocks or partitions. Each process is allocated according to the requirement.

In partition allocation, when there is more than one partion available to accomodate a process's request, a partition must be selected. To choose a particular partition, a partion allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation. Some partition allocation schemes are:

First Fit: The first partition which is of sufficient size is allocated.

Best Fit: The smallest partition which is sufficient is allocated.

Worst Fit: The largest partition is allocated.

EXPERIMENT NO. 6

Aim: Write a program to implement first fit, best fit and worst fit algorithm for memory management.

Program Code:

```
a) FIRST FIT
#include<bits/stdc++.h>
using namespace std;

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                break;
            }
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i+1 << "\t\t"
            << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
    firstFit(blockSize, m, processSize, n);
    return 0 ;
}

b) BEST FIT
#include<bits/stdc++.h>
using namespace std;

void bestFit(int blockSize[], int m, int processSize[], int n)
{
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));
```

```
    for (int i=0; i<n; i++) {
        int bestIdx = -1;
        for (int j=0; j<m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1)
                    bestIdx = j;
                else if (blockSize[bestIdx] > blockSize[j])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);
    bestFit(blockSize, m, processSize, n);
    return 0;
}

c) WORST FIT
#include<bits/stdc++.h>
using namespace std;

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));

    for (int i=0; i<n; i++) {
        int wstIdx = -1;
        for (int j=0; j<m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (wstIdx == -1)
                    wstIdx = j;
                else if (blockSize[wstIdx] < blockSize[j])
                    wstIdx = j;
            }
        }
        if (wstIdx != -1) {
            allocation[i] = wstIdx;
            blockSize[wstIdx] -= processSize[i];
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
```

```
        cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);
    worstFit(blockSize, m, processSize, n);
    return 0;
}
```

OUTPUT

a)

Process No.	Process Size	Block no.
1	212	2
2	417	5
3	112	2
4	426	Not Allocated

b)

Process No.	Process Size	Block no.
1	212	4
2	417	2
3	112	3
4	426	5

c)

Process No.	Process Size	Block no.
1	212	5
2	417	2
3	112	5
4	426	Not Allocated

DELTA Pg No
Date / /

EXPERIMENT 7

AIM: Write a program to implement reader/writer problem using semaphore.

Theory:

The readers - writers problem is used to manage synchronization so that there are no problems with the object data. For example, if two readers access the object at same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object, i.e., when a writer is accessing the object, no reader or writer may access it. This can be implemented using semaphores. Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations wait and signal which are used for process synchronization. The wait operation decrements the value of its argument S, if it is positive.

EXPERIMENT NO. 7

Aim: Write a program to implement reader/writer problem using semaphore

Program Code:

```
#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);

        return 0;
}
```

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
```

EXPERIMENT 8

Aim: Write a program to implement Banker's algorithm for deadlock avoidance.

Theory

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for pre-determined 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.

EXPERIMENT NO. 8

Aim: Write a program to implement Banker's algorithm for deadlock avoidance

Program Code:

```
#include<iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                int allot[P][R])
{
    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -
            //                 allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];

    // Function to calculate need matrix
    calculateNeed(need, maxm, allot);

    // Mark all processes as infinish
    bool finish[P] = {0};

    // To store safe sequence
    int safeSeq[P];

    // Make a copy of available resources
    int work[R];
    for (int i = 0; i < R ; i++)
        work[i] = avail[i];

    // While all processes are not finished
    // or system is not in safe state.
    int count = 0;
    while (count < P)
    {
```

```
        // Find a process which is not finish and
        // whose needs can be satisfied with current
        // work[] resources.
        bool found = false;
        for (int p = 0; p < P; p++)
        {
            // First check if a process is finished,
            // if no, go for next condition
            if (finish[p] == 0)
            {
                // Check if for all resources of
                // current P need is less
                // than work
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                // If all needs of p were satisfied.
                if (j == R)
                {
                    // Add the allocated resources of
                    // current P to the available/work
                    // resources i.e.free the resources
                    for (int k = 0 ; k < R ; k++)
                        work[k] += allot[p][k];

                    // Add this process to safe sequence.
                    safeSeq[count++] = p;

                    // Mark this p as finished
                    finish[p] = 1;

                    found = true;
                }
            }
        }

        // If we could not find a next process in safe
        // sequence.
        if (found == false)
        {
            cout << "System is not in safe state";
            return false;
        }
    }

    // If system is in safe state then
    // safe sequence will be as below
    cout << "System is in safe state.\nSafe"
        " sequence is: ";
    for (int i = 0; i < P ; i++)
        cout << safeSeq[i] << " ";

    return true;
}

// Driver code
int main()
{
    int processes[] = {0, 1, 2, 3, 4};
```

```
    // Available instances of resources
    int avail[] = {3, 3, 2};

    // Maximum R that can be allocated
    // to processes
    int maxm[][R] = {{7, 5, 3},
                     {3, 2, 2},
                     {9, 0, 2},
                     {2, 2, 2},
                     {4, 3, 3}};

    // Resources allocated to processes
    int allot[][R] = {{0, 1, 0},
                     {2, 0, 0},
                     {3, 0, 2},
                     {2, 1, 1},
                     {0, 0, 2}};

    // Check system is in safe state or not
    isSafe(processes, avail, maxm, allot);

    return 0;
}
```

OUTPUT

```
System is in safe state.
Safe sequence is: 1 3 4 0 2
```