# OPERATING SYSTEMS LAB

# ETCS 352

Submitted To:

Mr. Navdeep Bhora

Submitted by:

Name: Aditya Gupta

Serial No: 37

Enrolment No: 35115002717

Class: CSE-2(B)

**Maharaja Surajmal Institute of Technology,**
**C-4 Janak Puri, New Delhi 110058**

# INDEX

# EXPERIMENT - 1

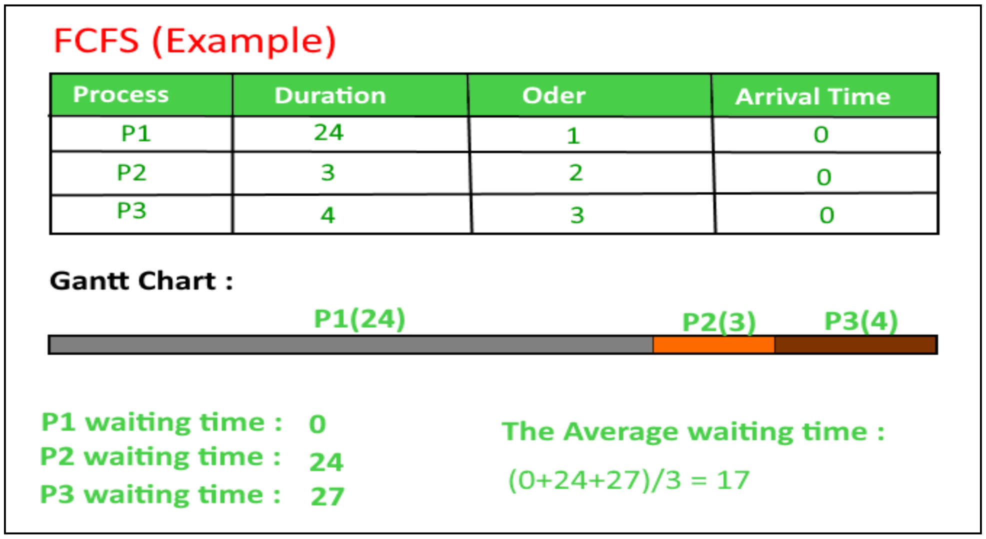
**AIM:** Write a program to implement CPU Scheduling for First Come First Serve.

## THEORY:

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.
First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

## IMPLEMENTATION:

1- Input the processes along with their burst time (bt).
2- Find waiting time (wt) for all processes.
3- As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e. wt[0] = 0.
4- Find **waiting time** for all other processes i.e. for
process i ->
wt[i] = bt[i-1] + wt[i-1] .
5- Find **turnaround time** = waiting_time + burst_time
for all processes.
6- Find **average waiting time** =
total_waiting_time /
no_of_processes. 7- Similarly, find **average turnaround time** =
total_turn_aroun_time / no_of_processes.


FCFS (Example)

| Process | Duration | Oder | Arrival Time |
|---|---|---|---|
| P1 | 24 | 1 | 0 |
| P2 | 3 | 2 | 0 |
| P3 | 4 | 3 | 0 |

Gantt Chart :

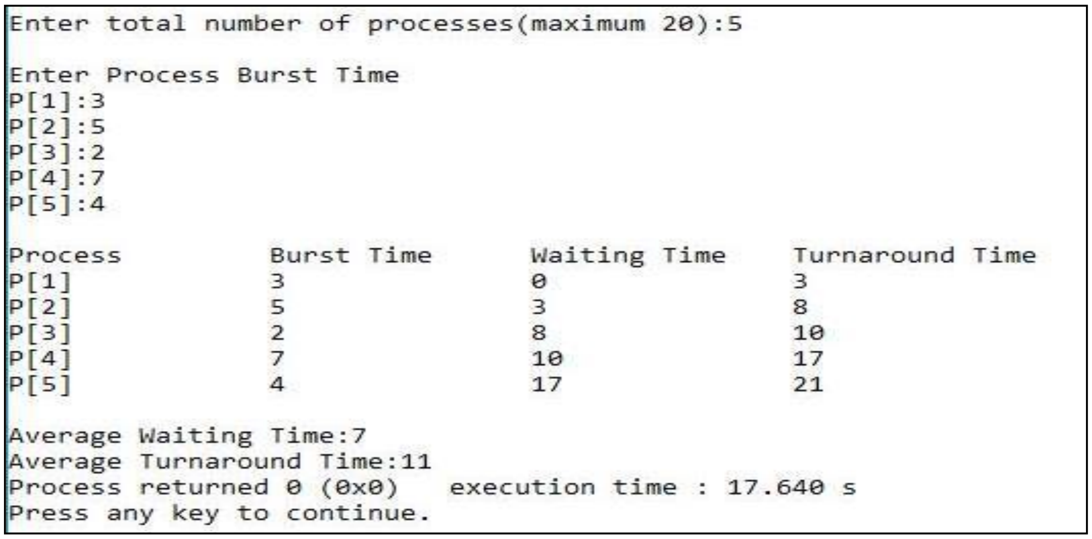
P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :
(0+24+27)/3 = 17

**CODE :**

```
#include<iostream>

using namespace std;

 int main() {

       int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j; cout<<"Enter
       total number of processes(maximum 20):"; cin>>n;

       cout<<"\nEnter Process Burst Time\n";
       for(i=0;i<n;i++) {

              cout<<"P["<<i+1<<"]:";

              cin>>bt[i];

       }

       wt[0]=0;        //waiting time for first process is
       0 for(i=1;i<n;i++) {

              wt[i]=0;
              for(j=0;j<i;j++)
              wt[i]+=bt[j];

       }

       cout<<"\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time";

       //calculating turnaround
       time for(i=0;i<n;i++) {

              tat[i]=bt[i]+wt[i];
              avwt+=wt[i];
              avtat+=tat[i];

              cout<<"\nP["<<i+1<<"]"<<"\t\t"<<bt[i]<<"\t\t"<<wt[i]<<"\t\t"<<tat[i];

       }

       avwt/=i;
       avtat/=i
       ;

       cout<<"\n Average Waiting Time:"<<avwt;
       cout<<"\n Average Turnaround Time:"<<avtat;
       return 0;

}
```

**OUTPUT:**



```
Enter total number of processes(maximum 20):5

Enter Process Burst Time
P[1]:3
P[2]:5
P[3]:2
P[4]:7
P[5]:4

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            3               0               3
P[2]            5               3               8
P[3]            2               8               10
P[4]            7               10              17
P[5]            4               17              21

Average Waiting Time:7
Average Turnaround Time:11
Process returned 0 (0x0)   execution time : 17.640 s
Press any key to continue.
```

# EXPERIMENT - 2

**AIM :** Write a program to implement CPU Scheduling for Shortest Job First.

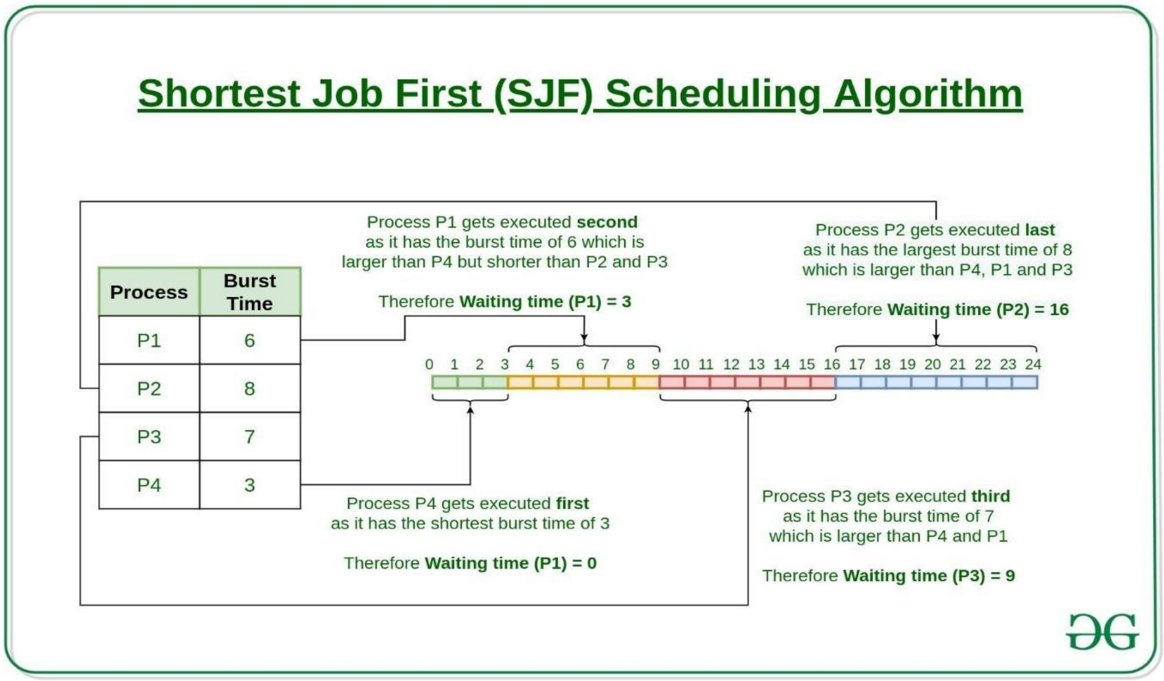
**THEORY :**

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJF is a non-preemptive algorithm.

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

**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 is having minimum Burst time.


Shortest Job First (SJF) Scheduling Algorithm
Process | Burst Time
P1 6
P2 8
P3 7
P4 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Process P1 gets executed second as it has the burst time of 6 which is larger than P4 but shorter than P2 and P3
Therefore Waiting time (P1) = 3
Process P2 gets executed last as it has the largest burst time of 8 which is larger than P4, P1 and P3
Therefore Waiting time (P2) = 16
Process P4 gets executed first as it has the shortest burst time of 3
Therefore Waiting time (P1) = 0
Process P3 gets executed third as it has the burst time of 7 which is larger than P4 and P1
Therefore Waiting time (P3) = 9

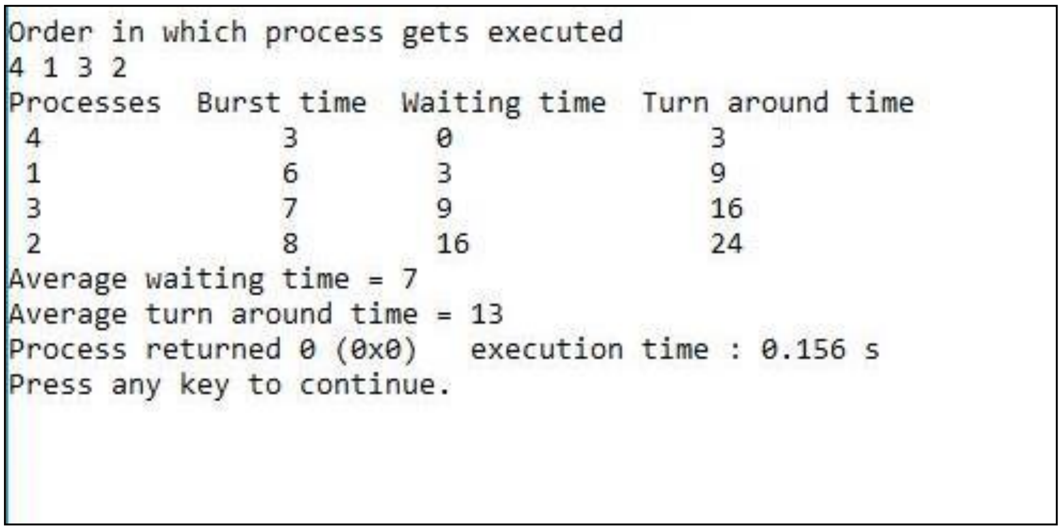
## CODE:

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

    int pid; // Process ID
    int bt;    // Burst Time

};


int comparison(Process a, Process b)
        { return (a.bt < b.bt);

}


// Function to find the waiting time for all processes
void findWaitingTime(Process proc[], int n, int wt[]) {

      // waiting time for first process is 0 wt[0] = 0;

      // calculating waiting time
      for (int i = 1; i < n ; i++ )

            wt[i] = proc[i-1].bt + wt[i-1] ;

}


// Function to calculate turn around time

void findTurnAroundTime(Process proc[], int n, int wt[], int tat[]) {

          bt[i] + wt[i];

         for (int i = 0; i < n ; i++)
             tat[i] = proc[i].bt + wt[i];

}


//Function to calculate average time

void findavgTime(Process proc[], int n) {

      int wt[n], tat[n], total_wt = 0, total_tat = 0;

//      Function to find waiting time of all
processes findWaitingTime(proc, n, wt);
```

```
//      Function to find turn around time for all
processes findTurnAroundTime(proc, n, wt, tat);




//      Display processes along with all details

        cout << "\nProcesses "<< " 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 << " " << proc[i].pid << "\t\t" << proc[i].bt << "\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;

}


// Driver code

// int main()

{

        Process proc[] = {{1, 6}, {2, 8}, {3, 7}, {4, 3}}; int n =
        sizeof proc / sizeof proc[0];

//          Sorting processes by burst time.
               sort(proc, proc + n,
               comparison);

          cout << "Order in which process gets
          executed\n"; for (int i = 0 ; i < n; i++)

                cout << proc[i].pid <<" ";
        findavgTime(proc, n);

                return 0;
```

}

**OUTPUT:**



```
Order in which process gets executed
4 1 3 2
Processes  Burst time  Waiting time  Turn around time
 4              3       0                3
 1              6       3                9
 3              7       9                16
 2              8       16               24
Average waiting time = 7
Average turn around time = 13
Process returned 0 (0x0)   execution time : 0.156 s
Press any key to continue.
```

## EXPERIMENT - 3

**AIM:** Write a program to perform Priority Scheduling.

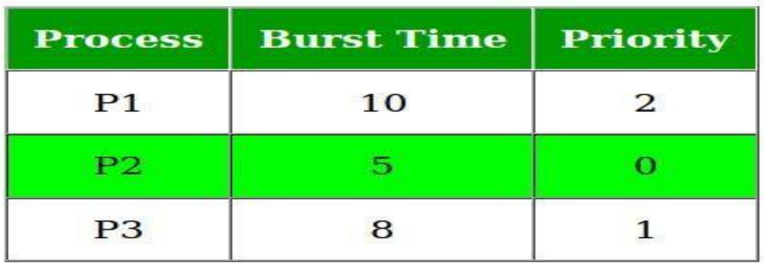
**THEORY:**

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. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

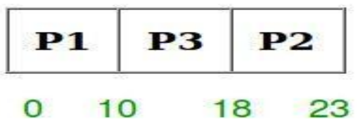
**IMPLEMENTATION :**

1- First input the processes with their burst time and priority.
2- Sort the processes, burst time and priority according to the priority.
3- Now simply apply FCFS algorithm.

**EXAMPLE:**



| Process | Burst Time | Priority |
|---|---|---|
| P1 | 10 | 2 |
| P2 | 5 | 0 |
| P3 | 8 | 1 |



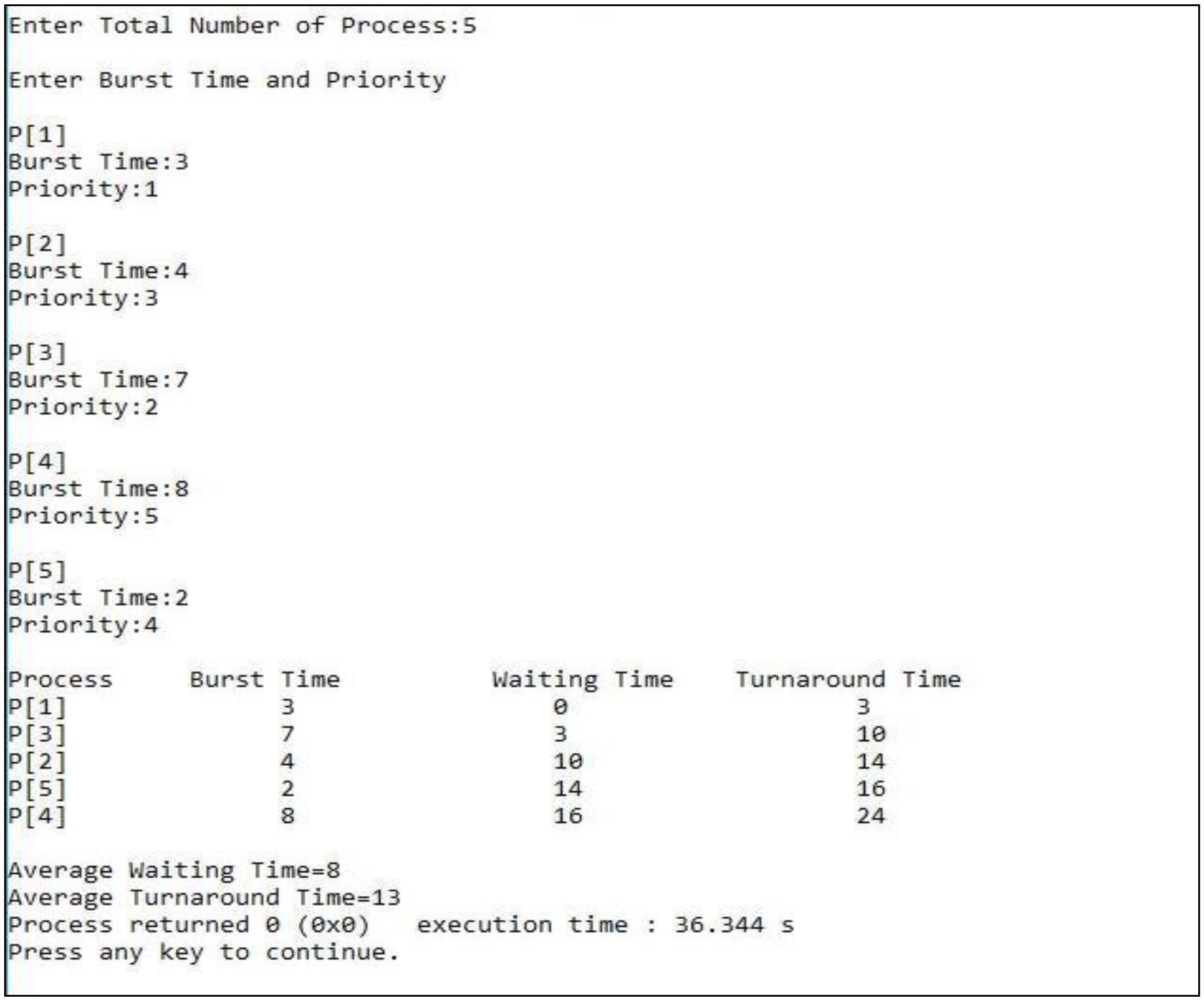
**CODE:**

```
#include<iostream>
  using namespace std;
  int main() {
      int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat; cout<<"Enter
      Total Number of Process:"; cin>>n;
      cout<<"\nEnter Burst Time and Priority\n";
```

```
 for(i=0;i<n;i++) {
     cout<<"\nP["<<i+1<<"]\n";



     cout<<"Burst Time:";
     cin>>bt[i];
     cout<<"Priority:";
     cin>>pr[i];
     p[i]=i+1;              //contains process number

 }

//sorting burst time, priority and process number in ascending order using selection sort
 for(i=0;i<n;i++) {
     pos=i;
     for(j=i+1;j<n;j++) {
         if(pr[j]<pr[pos])
             pos=j;
     }
     temp=pr[i];
     pr[i]=pr[pos];
     pr[pos]=temp;
     temp=bt[i];
     bt[i]=bt[pos];
     bt[pos]=temp;
     temp=p[i];
     p[i]=p[pos];
     p[pos]=temp;
 }
 wt[0]=0;
 for(i=1;i<n;i++)
 {
     wt[i]=0;
     for(j=0;j<i;j++)
         wt[i]+=bt[j];
     total+=wt[i];
 }

 avg_wt=total/n ;        //average waiting time
 total=0;
```

```
        cout<<"\nProcess\t      Burst Time        \tWaiting Time\tTurnaround Time";

        for(i=0;i<n;i++) {
            tat[i]=bt[i]+wt[i];        //calculate turnaround time
            total+=tat[i];
            cout<<"\nP["<<p[i]<<"]\t\t   "<<bt[i]<<"\t\t       "<<wt[i]<<"\t\t\t"<<tat[i];

        }
        avg_tat=total/n;        //average turnaround time

    cout<<"\n\nAverage Waiting Time="<<avg_wt;
    cout<<"\nAverage Turnaround Time="<<avg_tat;
    return 0;
}
```

**OUTPUT :**



```
Enter Total Number of Process:5

Enter Burst Time and Priority

P[1]
Burst Time:3
Priority:1

P[2]
Burst Time:4
Priority:3

P[3]
Burst Time:7
Priority:2

P[4]
Burst Time:8
Priority:5

P[5]
Burst Time:2
Priority:4

Process     Burst Time          Waiting Time    Turnaround Time
P[1]              3                 0                   3
P[3]              7                 3                   10
P[2]              4                 10                  14
P[5]              2                 14                  16
P[4]              8                 16                  24

Average Waiting Time=8
Average Turnaround Time=13
Process returned 0 (0x0)   execution time : 36.344 s
Press any key to continue.
```

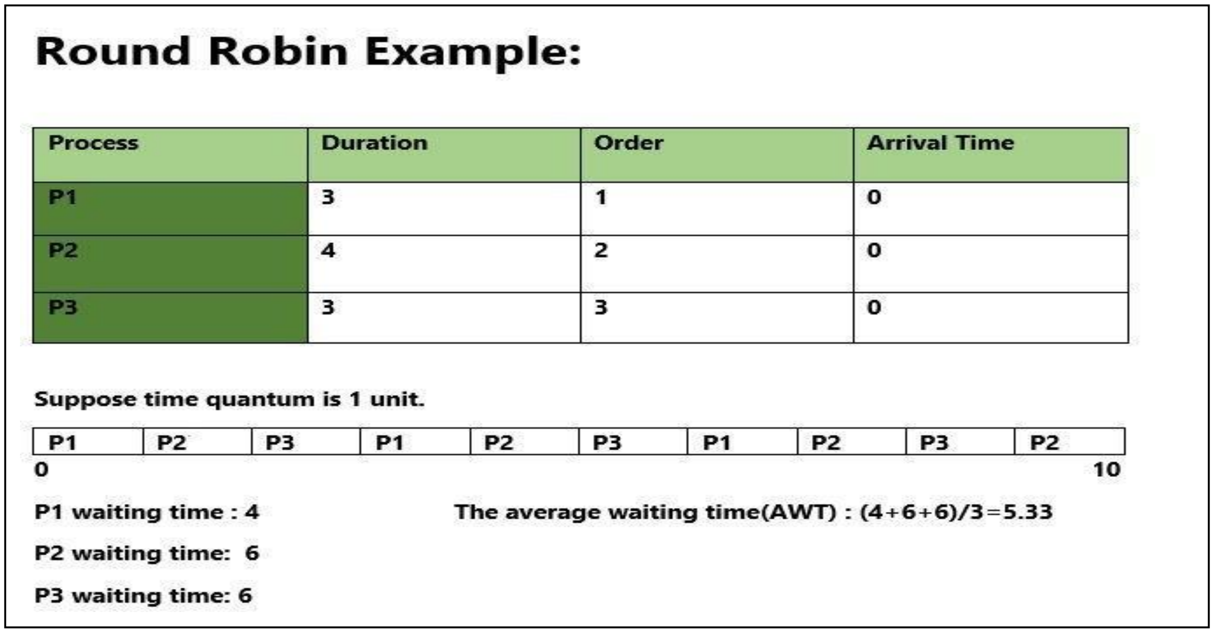
# EXPERIMENT-4

**AIM:** Write a program to illustrate Round Robin Scheduling algorithm.

## THEORY:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.



**Round Robin Example:**

| Process | Duration | Order | Arrival Time |
|---|---|---|---|
| P1 | 3 | 1 | 0 |
| P2 | 4 | 2 | 0 |
| P3 | 3 | 3 | 0 |

Suppose time quantum is 1 unit.

| P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 | P2 |
|---|---|---|---|---|---|---|---|---|---|

0 10

P1 waiting time : 4

P2 waiting time: 6

P3 waiting time: 6

The average waiting time(AWT) : (4+6+6)/3=5.33

## IMPLEMENTATION:

1- Create an array **rem_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)

2- Create another array **wt[]** to store waiting times of processes. Initialize this array as 0.

3- Initialize time : t = 0

4- Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.

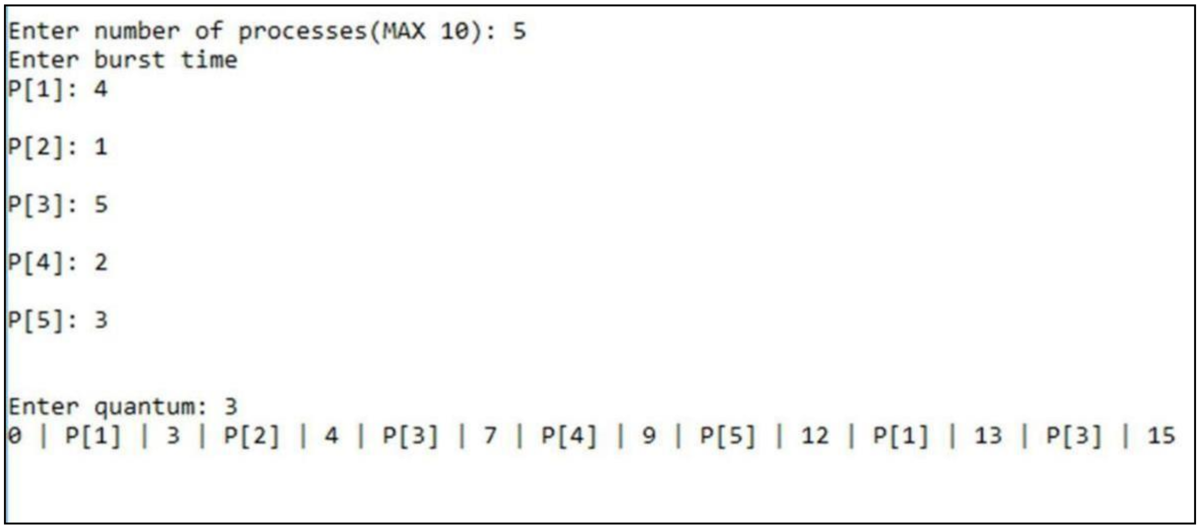
```
If rem_bt[i] > quantum
  (i)  t = t + quantum
```

```
        (ii) bt_rem[i] -= quantum;
    Else // Last cycle for this process
        (i)  t = t + bt_rem[i];
        (ii) wt[i] = t - bt[i]
        (ii) bt_rem[i] = 0; // This process is over
```

**CODE:**

```
#include<iostream>
#include<conio.h>
using namespace std;
int main() {
int wtime[10],btime[10],rtime[10],num,quantum,total;
cout<<"Enter number of processes(MAX 10): ";
cin>>num;
cout<<"Enter burst time";
for(int i=0;i<num;i++) {
cout<<"\nP["<<i+1<<"]: ";
    cin>>btime[i];
rtime[i] = btime[i];
wtime[i]=0;
}
cout<<"\n\nEnter quantum: "; cin>>quantum;
int rp = num;
int i=0;
int time=0;
cout<<"0"; wtime[0]=0;

while(rp!=0) { if(rtime[i]>quantum) {
    rtime[i]=rtime[i]-quantum;
cout<<" | P["<<i+1<<"] | ";
time+=quantum;
    cout<<time;
    }
  else if(rtime[i]<=quantum && rtime[i]>0) {
  time+=rtime[i];
rtime[i]=rtime[i]-rtime[i];
cout<<" | P["<<i+1<<"]";
   rp--;
   cout<<time;
  }
i++;
if(i==num)
        i=0;
}
getch();
return 0;
}
```

**OUTPUT:**



```
Enter number of processes(MAX 10): 5
Enter burst time
P[1]: 4

P[2]: 1

P[3]: 5

P[4]: 2

P[5]: 3


Enter quantum: 3
0 | P[1] | 3 | P[2] | 4 | P[3] | 7 | P[4] | 9 | P[5] | 12 | P[1] | 13 | P[3] | 15
```

# EXPERIMENT-5

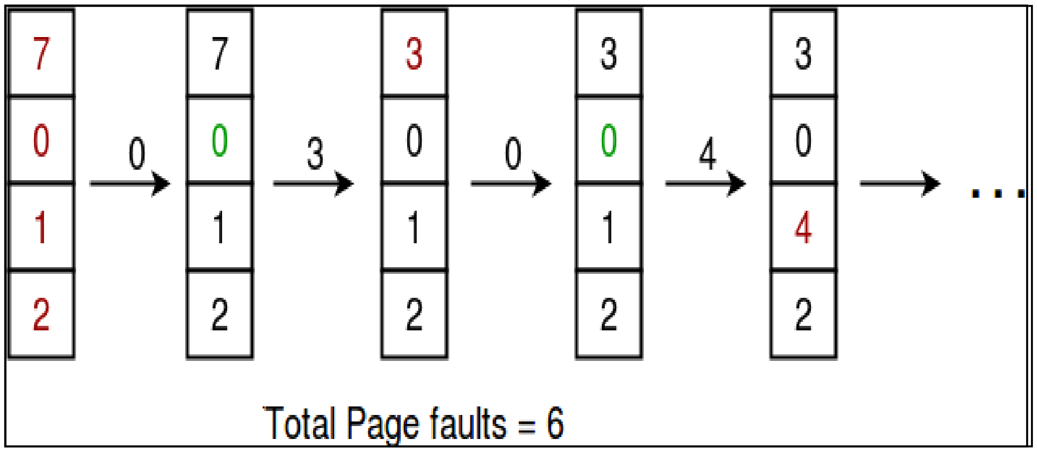
## AIM:

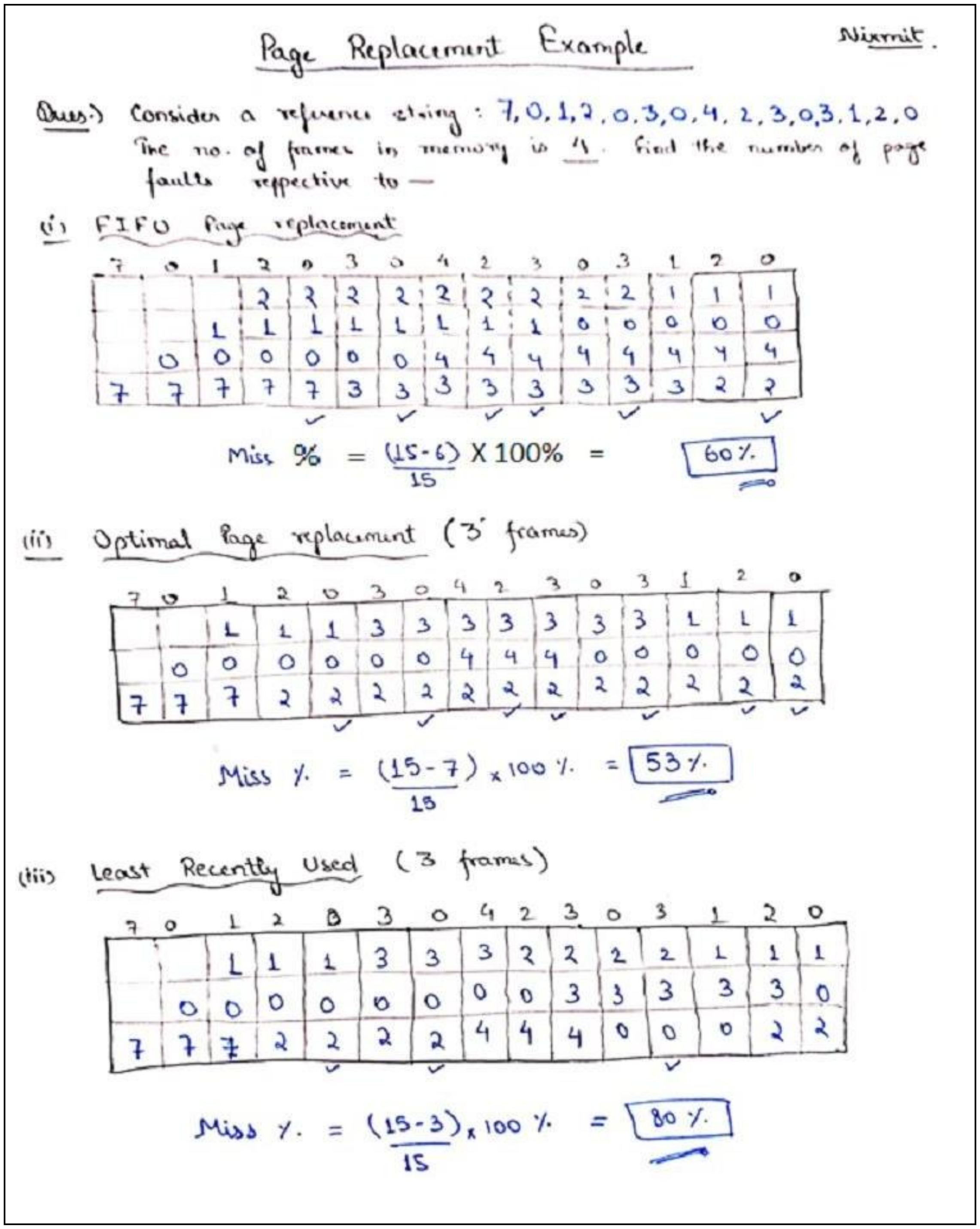
(a) Write a program to illustrate Least Recent Used Page replacement algorithm.

## THEORY:

In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely to be a hit.
Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already their so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.





# Page Replacement Example

Nirmit.

**Ques:)** Consider a reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0
The no. of frames in memory is 4. Find the number of page faults respective to —

(i) FIFO Page replacement

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 1 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
|  |  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 |
|  |  |  |  | ✓ |  | ✓ |  | ✓ | ✓ |  | ✓ |  |  | ✓ |

$$\text{Miss } \% = \frac{(15-6)}{15} \times 100\% = \boxed{60\%}$$

(ii) Optimal Page replacement (3 frames)

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 1 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|  |  |  |  | ✓ |  | ✓ |  | ✓ | ✓ |  | ✓ |  | ✓ | ✓ |

$$\text{Miss } \% = \frac{(15-7)}{15} \times 100\% = \boxed{53\%}$$

(iii) Least Recently Used (3 frames)

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 1 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | 1 | 1 | 1 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 0 |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 2 | 2 |
|  |  |  |  | ✓ |  | ✓ |  |  |  |  | ✓ |  |  |  |

$$\text{Miss } \% = \frac{(15-3)}{15} \times 100\% = \boxed{80\%}$$

## ALGORITHM:

Let **capacity** be the number of pages that memory can hold. Let **set** be the current set of pages in memory.

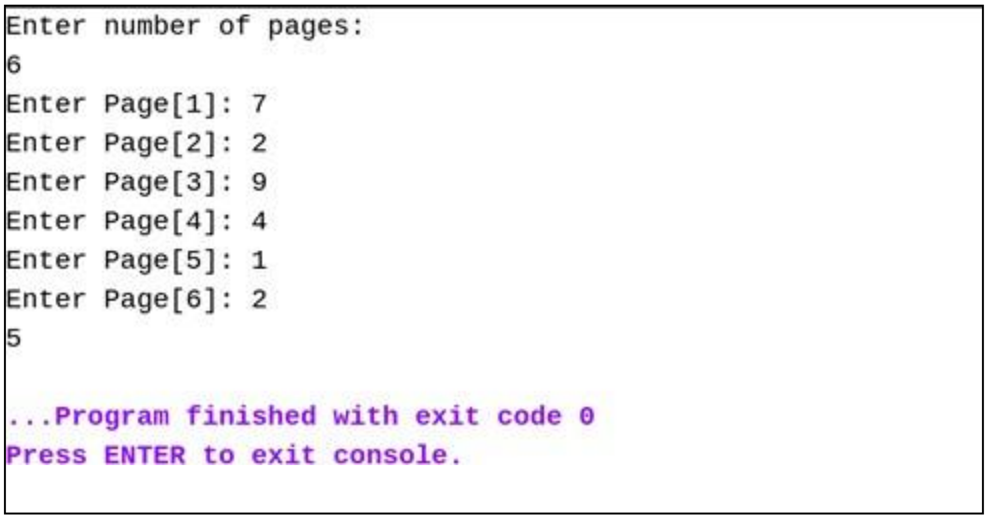
```
1- Start traversing the pages.
 i) If set holds less pages than capacity.
   a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
   b) Simultaneously maintain the recent occurred index of each page in a map called indexes.
   c) Increment page fault
 ii) Else
   If current page is present in set, do nothing.
   Else
     a) Find the page in the set that was least recently used. We find it using index array.

     We basically need to replace the page with minimum index.
     b) Replace the found page with current page.
     c) Increment page faults.
     d) Update index of current page.

2. Return page faults.
```

## CODE:

```
#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;
```

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

        return page_faults;
}

int main() {
        int n;
        cout<<"Enter number of pages:\n";
        cin>>n;
        int pages[n];
        int r,i;
        for(i=0;i<n;i++) {
          cout<<"Enter Page["<<i+1<<"]: ";
          cin>>r;
          pages[i] = r;
        }
        n = sizeof(pages)/sizeof(pages[0]);
        int capacity = 4;
        cout << pageFaults(pages, n, capacity);
        return 0;
}
```

**OUTPUT:**



```
Enter number of pages:
6
Enter Page[1]: 7
Enter Page[2]: 2
Enter Page[3]: 9
Enter Page[4]: 4
Enter Page[5]: 1
Enter Page[6]: 2
5

...Program finished with exit code 0
Press ENTER to exit console.
```

**(b)** Write a program to illustrate First In First Out Page replacement algorithm.

## THEORY:

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

**Belady's anomaly–** Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3,2, 1,0, 3, 2, 4, 3, 2, 1, 0,4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

## ALGORITHM:

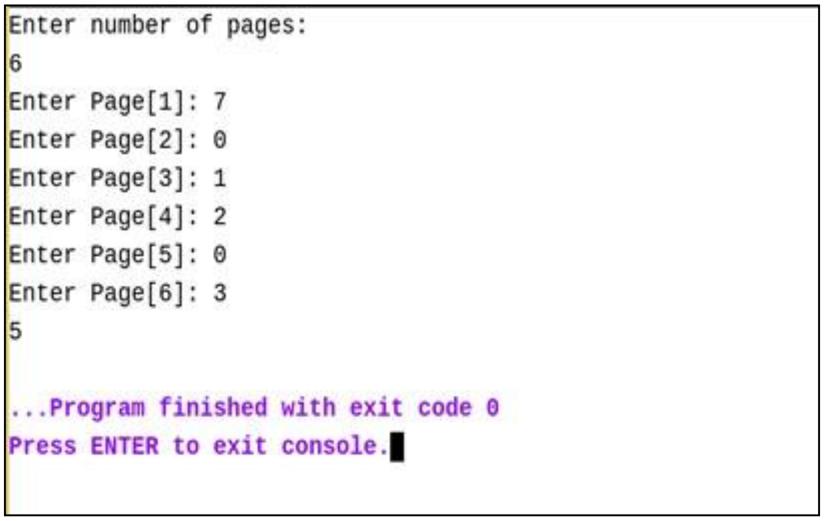
```
1. Start traversing the pages.
 i) If set holds less pages than capacity.
   a) Insert page into the set one by one until the size of set reaches capacity or all page requests
are processed.
   b) Simultaneously maintain the pages in the queue to perform FIFO.
   c) Increment page fault
 ii) Else
   If current page is present in set, do nothing.
   Else
     a) Remove the first page from the queue as it was the first to be entered in the memory.
     b) Replace the first page in the queue with the current page in the string.
     c) Store current page in the queue.
     d) Increment page faults.

2. Return page faults.
```

## CODE:

```
#include<bits/stdc++.h>
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++) {
                if (s.size() < capacity) {
                        if (s.find(pages[i])==s.end()) {
```

```
                        s.insert(pages[i]);
                        page_faults++;
                        indexes.push(pages[i]);
                    }
                }
                else {

                    if (s.find(pages[i]) == s.end()) {
                        int val = indexes.front();
                        indexes.pop();
                        s.erase(val);
                        s.insert(pages[i]);
                        indexes.push(pages[i]);
                        page_faults++;
                    }
                }
            }
            return page_faults;
}

int main() {
            int n;
            cout<<"Enter number of pages:\n";
            cin>>n;
            int pages[n];
            int r,i;
            for(i=0;i<n;i++) {
                cout<<"Enter Page["<<i+1<<"]: ";
                cin>>r;
                pages[i] = r;
            }
            n = sizeof(pages)/sizeof(pages[0]);
            int capacity = 4;
           cout << pageFaults(pages, n, capacity);
           return 0;
           }
```

## OUTPUT:



```
Enter number of pages:
6
Enter Page[1]: 7
Enter Page[2]: 0
Enter Page[3]: 1
Enter Page[4]: 2
Enter Page[5]: 0
Enter Page[6]: 3
5

...Program finished with exit code 0
Press ENTER to exit console.
```

**(c)** Write a program to illustrate Optimal Page replacement algorithm.

## THEORY/ ALGORITHM :

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.
In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

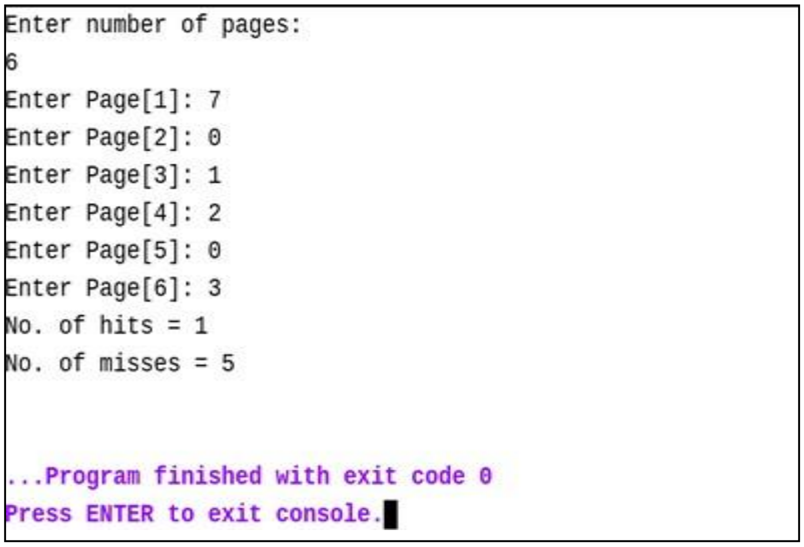
The idea is simple, for every reference we do following:

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

## CODE:

```
#include <bits/stdc++.h>
using namespace std;
bool search(int key, vector<int>& fr) {
        for (int i = 0; i < fr.size(); i++)
                if (fr[i] == key)
                        return true;
        return false;
}
int predict(int pg[], vector<int>& fr, int pn, int index) {
        int res = -1, farthest = index;
        for (int i = 0; i < fr.size(); i++) {
                int j;
                for (j = index; j < pn; j++) {
                        if (fr[i] == pg[j]) {
                                if (j > farthest) {
                                        farthest = j;
                                        res = i;
                                }
                                break;
                        }
                }
                if (j == pn)
                        return i;
        }
        return (res == -1) ? 0 : res;
}
void optimalPage(int pg[], int pn, int fn) {
	vector<int> fr;
	int hit = 0;
	for (int i = 0; i < pn; i++) {
		if (search(pg[i], fr)) {
			hit++;
			continue;
		}
		if (fr.size() < fn)
			fr.push_back(pg[i]);
		else {
			int j = predict(pg, fr, pn, i + 1);
			fr[j] = pg[i];
		}
	}
	cout << "No. of hits = " << hit << endl;
	cout << "No. of misses = " << pn - hit << endl;
}

int main() {
	int n;
	cout<<"Enter number of pages:\n";
	cin>>n;
	int pages[n];
	int r,i;
	for(i=0;i<n;i++) {
	    cout<<"Enter Page["<<i+1<<"]: ";
	    cin>>r;
	    pages[i] = r;
	}
	n = sizeof(pages)/sizeof(pages[0]);
	int capacity = 4;
	optimalPage(pages, n, capacity);
	return 0;
}
```

## OUTPUT:



```
Enter number of pages:
6
Enter Page[1]: 7
Enter Page[2]: 0
Enter Page[3]: 1
Enter Page[4]: 2
Enter Page[5]: 0
Enter Page[6]: 3
No. of hits = 1
No. of misses = 5


...Program finished with exit code 0
Press ENTER to exit console.
```

# EXPERIMENT-6

**AIM:** **a)** Write a program to illustrate First Fit Memory Management.

## THEORY:

In the first fit, the partition is allocated which is first sufficient from the top of Main Memory.

- Its advantage is that it is the fastest search as it searches only the first block i.e. enough to assign a process.
- It may have problems of not allowing processes to take space even if it was possible to allocate.

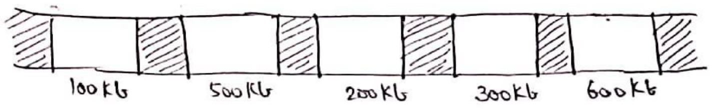
## ALGORITHM:

1- Input memory blocks with size and processes with size.
2- Initialize all memory blocks as free.
3- Start by picking each process and check if it can be assigned to current block.
4- If size-of-process <= size-of-block if yes then assign and check for next process.
5- If not then keep checking the further blocks.

# Memory Management Problem

Nirmit.

Q.) Given 5 memory partitions of 100 kb, 500 kb, 200 kb, 300 kb, 600 kb (in order), how would the first-fit, best-fit and worst-fit algorithms place processes of 212 kb, 417 kb, 112 kb and 426 kb (in order)? Which algorithm makes the most efficient use of memory?



→ let P1 = 212 kb
P2 = 417 kb
P3 = 112 kb
P4 = 426 kb

- First Fit :-

212 K is put in 500 K partition
417 K is put in 600 K partition
112 K is put in 288 K partition (500 K − 212 K)
426 K must wait

- Best fit :-

212 K is put in 300 K partition
417 K is put in 500 K partition
112 K is put in 200 K partition
426 K is put in 600 K partition

- Worst fit :-

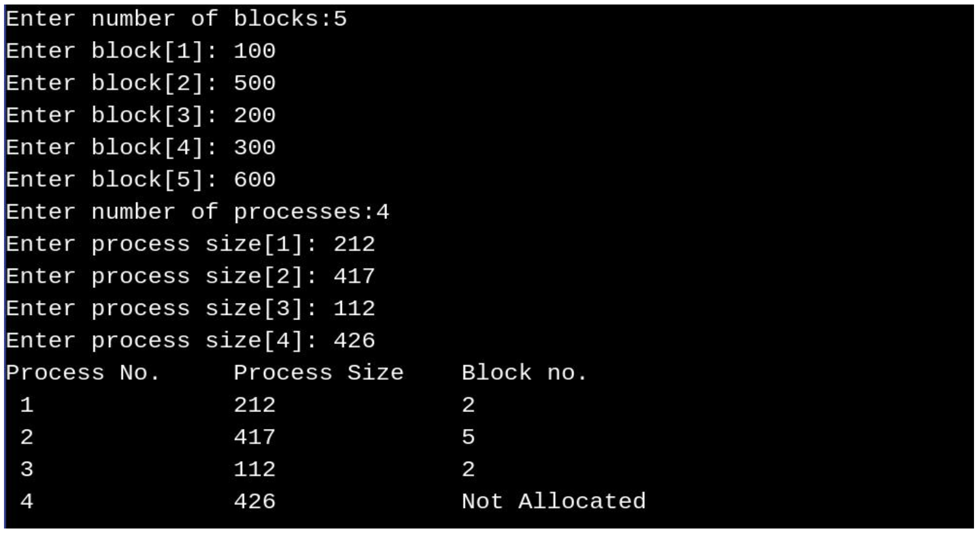
212 K is put in 600 K partition
417 K is put in 500 K partition
112 K is put in 388 K partition (600 K − 212 K)
426 K must wait

∴ In this example, Best-fit partition turns out to be the most efficient algorithm.

## CODE:

```
#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]) {
                                allocation[i] = j;
                                blockSize[j] -= processSize[i];
                                break;
                        }
                }
        }
        cout << "Process 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() {
   //{100, 500, 200, 300, 600};
        int n;
        cout<<"Enter number of blocks:";
        cin>>n;
        int blockSize[n];
        int r,i;
        for(i=0;i<n;i++) {
           cout<<"Enter block["<<i+1<<"]: ";
           cin>>blockSize[i];
        }
        cout<<"Enter number of processes:";
        //{212, 417, 112, 426};
        cin>>n;
        int processSize[n];
        for(i=0;i<n;i++) {
           cout<<"Enter process size["<<i+1<<"]: ";
           cin>>processSize[i];
        }
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        n = sizeof(processSize) / sizeof(processSize[0]);
        firstFit(blockSize, m, processSize, n);
        return 0 ;
}
```

## OUTPUT:



```
Enter number of blocks:5
Enter block[1]: 100
Enter block[2]: 500
Enter block[3]: 200
Enter block[4]: 300
Enter block[5]: 600
Enter number of processes:4
Enter process size[1]: 212
Enter process size[2]: 417
Enter process size[3]: 112
Enter process size[4]: 426
Process No.     Process Size    Block no.
 1              212             2
 2              417             5
 3              112             2
 4              426             Not Allocated
```

**b)** Write a program to illustrate Best Fit Memory Management.

## THEORY:

Best fit allocates the process to a partition which is the smallest sufficient partition among the free available partitions.

## EXAMPLE:

Input : blockSize[] = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};

Output:

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

## ALGORITHM:

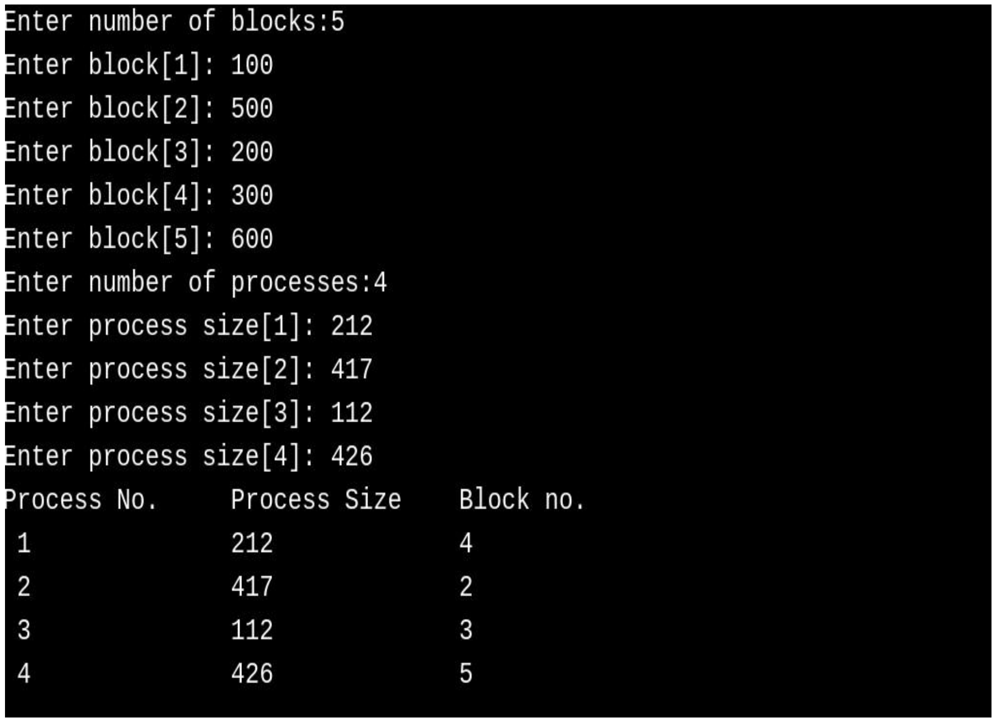
1- Input memory blocks and processes with sizes.
2- Initialize all memory blocks as free.
3- Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(bockSize[1], blockSize[2],.... blockSize[n]) > processSize[current], if found then assign it to the current process.
5- If not then leave that process and keep checking the further processes.

## CODE:

```
#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 << "Process 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 n;
        cout<<"Enter number of blocks:";
        cin>>n;
        int blockSize[n];
        int r,i;
        for(i=0;i<n;i++) {
           cout<<"Enter block["<<i+1<<"]: ";
           cin>>blockSize[i];
        }
        cout<<"Enter number of processes:";
        cin>>n;
        int processSize[n];
        for(i=0;i<n;i++) {
           cout<<"Enter process size["<<i+1<<"]: ";
           cin>>processSize[i];
        }
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        n = sizeof(processSize) / sizeof(processSize[0]);
        bestFit(blockSize, m, processSize, n);
        return 0 ;
}
```

## OUTPUT:



```
Enter number of blocks:5
Enter block[1]: 100
Enter block[2]: 500
Enter block[3]: 200
Enter block[4]: 300
Enter block[5]: 600
Enter number of processes:4
Enter process size[1]: 212
Enter process size[2]: 417
Enter process size[3]: 112
Enter process size[4]: 426
Process No.     Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
```

**c)** Write a program to illustrate Worst Fit Memory Management.

## THEORY:

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

## EXAMPLE:

Input: blockSize[] = {100, 500, 200, 300, 600};
processSize[] = {212, 417, 112, 426};

Output:

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

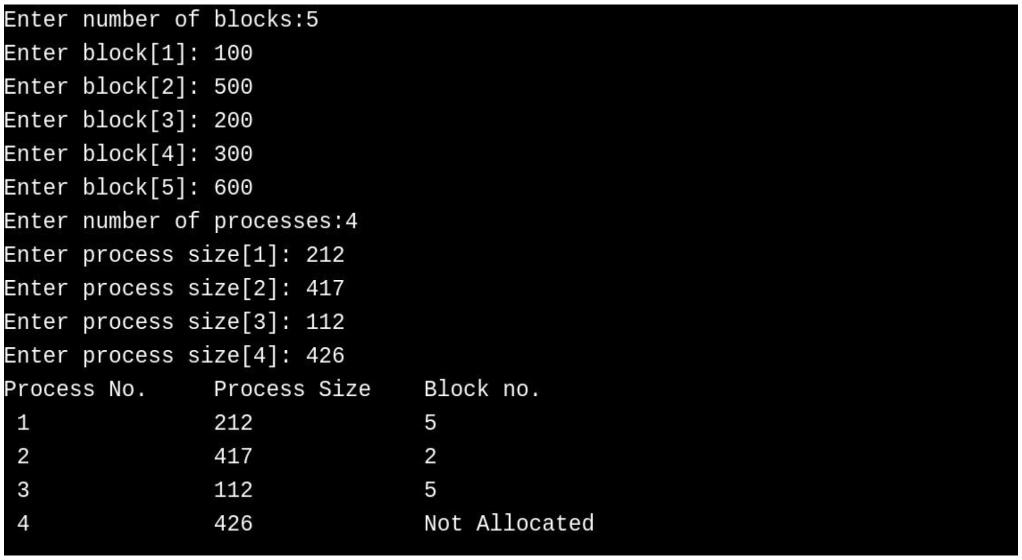
## ALGORITHM:

1- Input memory blocks and processes with sizes.
2- Initialize all memory blocks as free.
3- Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],. ...blockSize[n]) > processSize[current], if found then assign it to the current process.
4- If not then leave that process and keep checking the further processes.

## CODE:

```
#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 << "Process 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 n;
        cout<<"Enter number of blocks:";
        cin>>n;
        int blockSize[n];
        int r,i;
        for(i=0;i<n;i++) {
            cout<<"Enter block["<<i+1<<"]: ";
            cin>>blockSize[i];
        }
        cout<<"Enter number of processes:";
        cin>>n;
        int processSize[n];
        for(i=0;i<n;i++) {
            cout<<"Enter process size["<<i+1<<"]: ";
            cin>>processSize[i];
        }
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        n = sizeof(processSize) / sizeof(processSize[0]);
        worstFit(blockSize, m, processSize, n);
        return 0 ;
}
```

## OUTPUT:



```
Enter number of blocks:5
Enter block[1]: 100
Enter block[2]: 500
Enter block[3]: 200
Enter block[4]: 300
Enter block[5]: 600
Enter number of processes:4
Enter process size[1]: 212
Enter process size[2]: 417
Enter process size[3]: 112
Enter process size[4]: 426
Process No.     Process Size    Block no.
 1              212             5
 2              417             2
 3              112             5
 4              426             Not Allocated
```

# EXPERIMENT – 7

**AIM:** Write a program to illustrate Reader-Writer problems using semaphores.

## THEORY:

Problem Statement –
In an Operating System, we deal with various processes and these processes may use files that are present in the system. Basically, we perform two operations on a file i.e. read and write. All these processes can perform these two operations. But the problem that arises here is that:

- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

Solution –

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 the same time there is no problem. However if two writers or a reader and writer access the object at the 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. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows:

### Reader Process

"**mutex**" and "**wrt**" are semaphores that are initialized to 1. Also, "**rc**" is a variable that is initialized to 0. The **mutex** semaphore ensures mutual exclusion and **wrt** handles the writing mechanism and is common to the reader and writer process code.

The variable **rc** denotes the number of readers accessing the object. As soon as **rc** becomes 1, wait operation is used on **wrt**. This means that a writer cannot access the object anymore. After the read operation is done, **rc** is decremented. When **rc** becomes 0, signal operation is used on **wrt**. So a writer can access the object now.

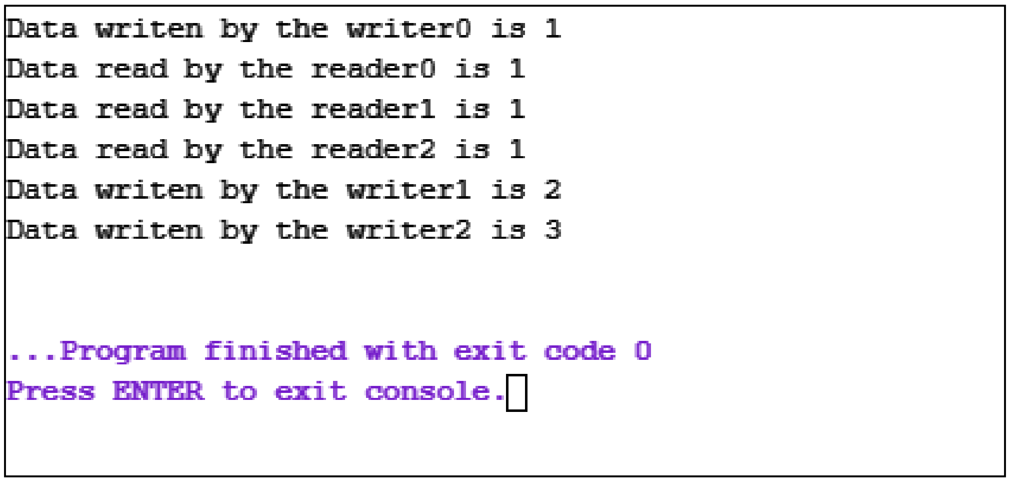
**Writer Process**

If a writer wants to access the object, wait operation is performed on **wrt**. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on **wrt**.

## CODE:

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
sem_t mutex,writeblock;
int data = 0,rcount = 0;

void *reader(void *arg) {
  int f;
  f = ((int)arg);
  sem_wait(&mutex);
  rcount = rcount + 1;
  if(rcount==1)
  sem_wait(&writeblock);
  sem_post(&mutex);
  cout<<"Data read by the reader"<<f<<" is "<<data<<endl;
  sleep(1);
  sem_wait(&mutex);
  rcount = rcount - 1;
  if(rcount==0)
  sem_post(&writeblock);
  sem_post(&mutex);
}

void *writer(void *arg) {
  int f;
  f = ((int) arg);
  sem_wait(&writeblock);
  data++;
  cout<<"Data writen by the writer"<<f<<" is "<<data<<endl;
  sleep(1);
  sem_post(&writeblock);
}

int main() {
  int i,b;
  pthread_t rtid[5],wtid[5];
  sem_init(&mutex,0,1);
  sem_init(&writeblock,0,1);
  for(i=0;i<=2;i++) {
    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++) {
    pthread_join(wtid[i],NULL);
    pthread_join(rtid[i],NULL);
  }
  return 0;
}
```

**OUTPUT:**



```
Data writen by the writer0 is 1
Data read by the reader0 is 1
Data read by the reader1 is 1
Data read by the reader2 is 1
Data writen by the writer1 is 2
Data writen by the writer2 is 3


...Program finished with exit code 0
Press ENTER to exit console.
```

# EXPERIMENT-8

**AIM:** Write a program to implement Banker's Algorithm for deadlock avoidance.

## THEORY:

Let 'n' be the number of processes in the system and 'm' be the number of resources types.
Available:

- It is a 1-d array of size 'm' indicating the number of available resources of each type.
- Available[ j ] = k means there are 'k' instances of resource type Rj

Max:

- It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[ i, j ] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation:

- It is a 2-d array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
- Allocation[ i, j ] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need:

- It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.
- Need [ i, j ] = k means process Pi currently allocated 'k' instances of resource type Rj
- Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

Allocation i specifies the resources currently allocated to process Pi and Need i specifies the additional resources that process Pi may still request to complete its task. Banker's algorithm consist of Safety algorithm and Resource request algorithm.

**Safety Algorithm:** The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length 'm' and 'n' respectively. Initialize: Work= Available Finish [i]=false; for i=1,2,……,n
2. Find an i such that both a) Finish [i]=false b) Need_i<=work if no such i exists goto step (4)
3. Work=Work + Allocation_i Finish[i]= true goto step(2)
4. If Finish[i]=true for all i, then the system is in safe state.

Safe sequence is the sequence in which the processes can be safely executed. In this post, implementation of Safety algorithm of Banker's Algorithm is done.

# Banker's Algorithm Example

Nirmit.

Ques.) Consider the system with resources A, B, C & D having 10, 13, 9 and 12 instances.

**Allocation**

|  | A | B | C | D |
|---|---|---|---|---|
| $P_0$ | 0 | 0 | 1 | 2 |
| $P_1$ | 3 | 1 | 2 | 1 |
| $P_2$ | 2 | 1 | 0 | 3 |
| $P_3$ | 1 | 3 | 1 | 2 |
| $P_4$ | 1 | 4 | 3 | 2 |
| (+) | 7 | 9 | 7 | 10 |

**Max Need**

|  | A | B | C | D |
|---|---|---|---|---|
| $P_0$ | 3 | 3 | 1 | 2 |
| $P_1$ | 4 | 2 | 5 | 2 |
| $P_2$ | 3 | 4 | 1 | 6 |
| $P_3$ | 2 | 3 | 2 | 4 |
| $P_4$ | 3 | 5 | 6 | 5 |

(a) What is the content of the matrix Available?

→

|  |  |  |  |  |
|---|---|---|---|---|
| Total | 10 | 13 | 9 | 12 |
| Allocated | 7 | 9 | 7 | 10 |
| (−) Available | 3 | 4 | 2 | 2 |

(b) What is the content of the matrix Need?

→ Need = Max − Allocation

|  | A | B | C | D |
|---|---|---|---|---|
| $P_0$ | 3 | 3 | 0 | 0 |
| $P_1$ | 1 | 1 | 3 | 1 |
| $P_2$ | 1 | 3 | 1 | 3 |
| $P_3$ | 1 | 0 | 1 | 2 |
| $P_4$ | 2 | 1 | 3 | 3 |

(c) Is this state safe? Show this by using safety algorithm.

→ Available : 3 4 2 2

**Allocation**

|  | A | B | C | D |
|---|---|---|---|---|
| $P_0$ | 0 | 0 | 1 | 2 |
| $P_1$ | 3 | 1 | 2 | 1 |
| $P_2$ | 2 | 1 | 0 | 3 |
| $P_3$ | 1 | 3 | 1 | 2 |
| $P_4$ | 1 | 4 | 3 | 2 |

**Need**

|  | A | B | C | D |
|---|---|---|---|---|
| $P_0$ | 3 | 3 | 0 | 0 |
| $P_1$ | 1 | 1 | 3 | 1 |
| $P_2$ | 1 | 3 | 1 | 3 |
| $P_3$ | 1 | 0 | 1 | 2 |
| $P_4$ | 2 | 1 | 3 | 3 |

If , Need < Available then, Work = Available + (Allocation);

| | Need < Available | Work |
|---|---|---|
| $P_0$ | (3 3 0 0) < (3 4 2 2) | ∴ Work : (3 4 2 2) + (0 0 1 2) = 3,4,3,4 |
| $P_3$ | (1 0 1 2) < (3 4 3 4) | ∴ Work : (3 4 3 4) + (1 3 1 2) = 4,7,4,6 |
| $P_4$ | (2 1 3 3) < (4 7 4 6) | ∴ Work : (4 7 4 6) + (1 4 3 2) = 5,11, 7, 8 |
| $P_1$ | (1 1 3 1) < (5 11 7 8) | ∴ Work : (5 11 7 8) + (3 1 2 1) = 8, 12, 9, 9 |
| $P_2$ | (1 3 1 3) < (8 12 9 9) | ∴ Work : (8 12 9 9) + (2 1 0 3) = 10, 13, 9, 12 |

∴ Sequence is $\boxed{P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2}$

(d) If a request from process $P_1$ arrives for (1 1 0 1), what will be the system's state?

⟶ Request : P1 ( 1 1 0 1)

$\therefore (\text{Allocation})_{P1} = (\text{Allocation})_{P1} + (\text{Request})_{P1}$

$= (3\ 1\ 2\ 1) + (1\ 1\ 0\ 1)$

$= 4\ 2\ 2\ 2$

& $\text{Available} = \text{Available} - (\text{Request})_{P1}$

$= (3\ 4\ 2\ 2) - (1\ 1\ 0\ 1)$

$= 2\ 3\ 2\ 1$

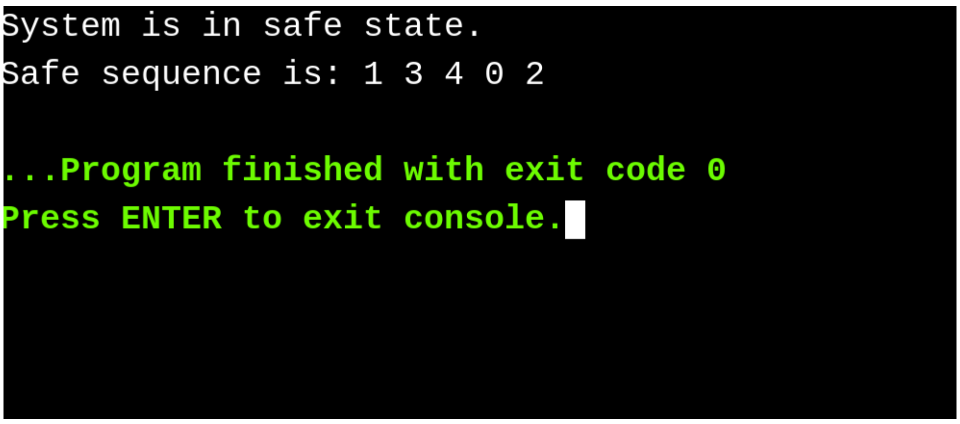
| | Allocation | | | | Max | Need |
|---|---|---|---|---|---|---|
| | A | B | C | D | A B C D | A B C D |
| $P_0$ | 0 | 0 | 1 | 2 | 3 3 1 2 | 3 3 0 2 |
| $P_1$ | 4 | 2 | 2 | 2 | 4 2 5 2 | 0 0 3 0 |
| $P_2$ | 2 | 1 | 0 | 3 | 3 4 1 6 | 1 3 1 3 |
| $P_3$ | 1 | 3 | 1 | 2 | 2 3 2 4 | 1 0 1 2 |
| $P_4$ | 1 | 4 | 3 | 2 | 3 5 6 5 | 2 1 3 3 |

Here, no need is less than available.
So, the request $P_1$( 1 1 0 1) is not granted.

## CODE:

```
#include<iostream>

using namespace std;

const int P = 5;
const int R = 3;
void calculateNeed(int need[P][R], int maxm[P][R], int allot[P][R])
{
         for (int i = 0 ; i < P ; i++)
                 for (int j = 0 ; j < R ; j++)
                          need[i][j] = maxm[i][j] - allot[i][j];
}
bool isSafe(int processes[], int avail[], int maxm[][R], int allot[][R]) {
         int need[P][R];
        calculateNeed(need, maxm, allot); bool finish[P] = {0};
        int safeSeq[P]; int work[R];
        for (int i = 0; i < R ; i++)
                 work[i] = avail[i]; int count = 0;
        while (count < P) {
                          bool found = false;
        for (int p = 0; p < P; p++)
        {
                 if (finish[p] == 0)
                 {
                          int j;
                  for (j = 0; j < R; j++)
                          if (need[p][j] > work[j])
                           break;
                      if (j == R)
                  {
                          for (int k = 0 ; k < R ; k++)
                                   work[k] += allot[p][k];
                           safeSeq[count++] = p;
                          finish[p] = 1;
                          found = true;
                    }
                 }

                 }

                 if (found == false) {
                          cout << "System is not in safe state"; return false;
                           }
                  }

cout << "System is in safe state.\nSafe sequence is: "; for (int i = 0; i < P ; i++)
cout << safeSeq[i] << " ";
return true;
}


int main() {
int processes[] = {0, 1, 2, 3, 4};
int avail[] = {3, 3, 2};
int maxm[][R] =
{
{7, 5, 3}, {3, 2, 2},{9, 0, 2}, {2, 2, 2},{4, 3, 3}
 };
int allot[][R] =
{
{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1},{0, 0, 2}
 };
        isSafe(processes, avail, maxm, allot); return 0;
}
```

**OUTPUT**



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

...Program finished with exit code 0
Press ENTER to exit console.
```