# INDEX

| S.no | Practical | Signature |
|---|---|---|
| 1 | Perform First Come First Serve Scheduling Algorithm. | |
| 2 | Perform Shortest Job First Scheduling Algorithm. | |
| 3 | Perform Preemptive Shortest Job First Scheduling Algorithm. | |
| 4 | Perform Round Robin Scheduling Algorithm. | |
| 5 | Perform Priority Scheduling Algorithm. | |
| 6 | Perform Preemptive Priority Scheduling Algorithm | |
| 7 | Perform Reader Writer Synchronisation Algorithm. | |
| 8 | Perform Dining Philosophers Synchronization Algorithm | |
| 9 | Perform bounded buffer Synchronization Algorithm | |
| 10 | Perform Banker's Algorithm | |
| 11 | Perform First Fit, Best Fit, and Worst Fit Memory Management Algorithms | |
| 12 | Perform First In First Out Page Replacement Algorithm | |
| 13 | Perform Least Recently Used Page Replacement Algorithm | |
| 14 | Perform Optimal Page Replacement Algorithm | |
| 15 | Perform First Come First Disk Scheduling Algorithm. | |
| 16 | Perform SCAN Disk Scheduling Algorithm. | |
| 17 | Perform C-SCAN Disk Scheduling Algorithm. | |

# PRACTICAL – 1

**AIM:** Perform First Come First Serve Scheduling Algorithm.

```
#include <bits/stdc++.h>
using namespace std;
bool sortpair(pair<int, int> process1, pair<int, int> process2)
{
 return process1.first < process2.first;
}
void firstComeFirstServe(int at[], int bt[], int n){
    vector<pair<int, int>> ans;
    for (int i = 0; i < n; i++){
        pair<int, int> process;
        process.first = at[i];
        process.second = bt[i];
        ans.push_back(p);
    }
    float avg = 0;
    sort(ans.begin(), ans.end(), sortpair);
    int wt[n], ct[n], st[n];
    wt[0] = 0, st[0] = 0, ct[0] = ans[0].second;
    for (int i = 1; i < n; i++){
        st[i] = ct[i - 1];
        wt[i] = st[i] - ans[i].first;
        if (wt[i] < 0) {
            wt[i] = 0;
        }
        ct[i] = st[i] + ans[i].second;
        avg += wt[i];
    }
    cout << "process schedule" << endl;
    for (int i = 0; i < n; i++){
     cout << i << " start time: " << st[i] << "s completion time: " << ct[i] << "s waiting
time :" << wt[i] <<endl ;
  }
cout << "average waiting time : " << avg / n <<"s"<< endl;
}
int main(){
    int at[] = {3, 2, 7, 7, 1};
```

```
    int bt[] = {4, 1, 3, 8, 2};
    int n = 5;
    firstComeFirstServe(at, bt, 5);
    return 0;
}
```

Output

```
process schedule
0 start time: 0s completion time: 2s waiting time :0
1 start time: 2s completion time: 3s waiting time :0
2 start time: 3s completion time: 7s waiting time :0
3 start time: 7s completion time: 10s waiting time :0
4 start time: 10s completion time: 18s waiting time :3
average waiting time : 0.6s

Process returned 0 (0x0)   execution time : 0.067 s
Press any key to continue.
```

# PRACTICAL - 2

**AIM:** Perform Shortest Job First Scheduling Algorithm.

```
#include <bits/stdc++.h>
using namespace std;
struct Process {
  int pid;
  int bt;
  int art;
};
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 findWaitingTime(Process proc[], int n, int wt[]) {
  int rt[n];
  for (int i = 0; i < n; i++)
  rt[i] = proc[i].bt;
  int complete = 0, t = 0, minm = INT_MAX;
  int shortest = 0, finish_time;
  bool check = false;
  while (complete != n) {
    for (int j = 0; j < n; j++) {
      if ((proc[j].art <= t) && (rt[j] < minm) && rt[j] > 0) {
        minm = rt[j];
        shortest = j;
        check = true;
      }
    }
    if (check == false) {
      t++;
      continue;
    }
    rt[shortest]--;
    minm = rt[shortest];
    if (minm == 0)
      minm = INT_MAX;
      if (rt[shortest] == 0) {
        complete++;
        check = false;
        finish_time = t + 1;
```

```
            wt[shortest] = finish_time -
            proc[shortest].bt -
            proc[shortest].art;
            if (wt[shortest] < 0)
               wt[shortest] = 0;
         }
         t++;
   }
}
void findavgTime(Process proc[], int n) {
   int wt[n], tat[n], total_wt = 0,
   total_tat = 0;
   findWaitingTime(proc, n, wt);
   findTurnAroundTime(proc, n, wt, tat);
   cout << "Processes " << " Burst time " << " Waiting time " << " Turn around
time\n";
   for (int i = 0; i < n; i++) {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      cout << " " << proc[i].pid << "\t\t" << proc[i].bt << "\t\t " << wt[i] << "\t\t " << tat[i]
<< endl;
   }
   cout << "\nAverage waiting time = " << (float)total_wt / (float)n; cout << "\nAverage
turn around time = " << (float)total_tat / (float)n;
}
int main() {
   Process proc[] = { { 1, 5, 1 }, { 2, 3, 1 }, { 3, 6, 2 }, { 4, 5, 3 } };
   int n = sizeof(proc) / sizeof(proc[0]);
   findavgTime(proc, n);
   return 0;
}
```

Output

```
Processes  Burst time  Waiting time  Turn around time
 1              5               3               8
 2              3               0               3
 3              6               12              18
 4              5               6               11

Average waiting time = 5.25
Average turn around time = 10
Process returned 0 (0x0)   execution time : 0.057 s
Press any key to continue.
```

# PRACTICAL - 3

**AIM:** Perform Preemptive Shortest Job First Scheduling Algorithm.

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

void findWaitingTime(int processes[][3], int n, int wt[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = processes[i][1];

    int complete = 0, t = 0, minm = INT_MAX, short_index = 0;
    bool check = false;

    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if (processes[j][2] <= t && rt[j] < minm && rt[j] > 0) {
                minm = rt[j];
                short_index = j;
                check = true;
            }
        }

        if (!check) {
            t++;
            continue;
        }

        rt[short_index]--;

        minm = rt[short_index];
        if (minm == 0)
            minm = INT_MAX;

        if (rt[short_index] == 0) {
            complete++;
            check = false;
            int fint = t + 1;
            wt[short_index] = (fint - processes[short_index][1] - processes[short_index][2]);
```

```
            if (wt[short_index] < 0)
                wt[short_index] = 0;
        }
        t++;
    }
}

void findTurnAroundTime(int processes[][3], int n, int wt[], int tat[]) {
    for (int i = 0; i < n; i++)
        tat[i] = processes[i][1] + wt[i];
}

void findavgTime(int processes[][3], int n) {
    int wt[n], tat[n];
    findWaitingTime(processes, n, wt);
    findTurnAroundTime(processes, n, wt, tat);

    cout << "Processes Burst Time Waiting Time Turn-Around Time" << endl;

    int total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        cout << " " << processes[i][0] << "\t\t" << processes[i][1] << "\t\t\t" << wt[i] <<
"\t\t\t" << tat[i] << endl;
    }

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

int main() {
    int proc[][3] = {{1, 6, 1}, {2, 8, 1}, {3, 7, 2}, {4, 3, 3}};
    int n = sizeof(proc) / sizeof(proc[0]);
    findavgTime(proc, n);
    return 0;
}
```

Output

```
Processes Burst Time     Waiting Time    Turn-Around Time
  1          6                3               9
  2          8                16                  24
  3          7                8               15
  4          3                0               3

Average waiting time = 6.75000
Average turn around time =  12.75
>
```

# PRACTICAL - 4

**AIM:** Perform Round Robin Scheduling Algorithm.

```
#include<iostream>
using namespace std;
void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum)
{
int rem_bt[n];
for (int i = 0 ; i < n ; i++)
rem_bt[i] = bt[i];
int t = 0;
while (1)
{
bool done = true;
for (int i = 0 ; i < n; i++)
{
if (rem_bt[i] > 0)
{
done = false;
if (rem_bt[i] > quantum)
{
t += quantum;
rem_bt[i] -= quantum;
}
else
{
t = t + rem_bt[i];
wt[i] = t - bt[i];
rem_bt[i] = 0;
}
}
}
if (done == true)
break;
}
}
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])
{
for (int i = 0; i < n ; i++)
```

```
tat[i] = bt[i] + wt[i];
}
void findavgTime(int processes[], int n, int bt[], int quantum)
{
int wt[n], tat[n], total_wt = 0, total_tat = 0;
findWaitingTime(processes, n, bt, wt, quantum);
findTurnAroundTime(processes, n, bt, wt, tat);
cout << "PN\t "<< " BT \t"<< " WT \t" << " TAT\n";
for (int i=0; i<n; i++)
{
total_wt = total_wt + wt[i];
total_tat = total_tat + tat[i];
cout << " " << i+1 << "\t" << bt[i] <<"\t "<< wt[i] <<"\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[] = {12, 7, 11};
int quantum = 3;
findavgTime(processes, n, burst_time, quantum);
return 0;
}
```

Output

```
PN         BT      WT       TAT
 1         12      16       28
 2         7       15       22
 3         11      19       30
Average waiting time = 16.6667
Average turn around time = 26.6667
Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.
```

# PRACTICAL - 5

**AIM:** Perform Priority Scheduling Algorithm.

```
#include <stdio.h>
void swap(int *a,int *b)
{
 int temp=*a;
 *a=*b;
 *b=temp;
}
int main()
{
 int n;
 printf("Enter Number of Processes: ");
 scanf("%d",&n);
 int b[n],p[n],index[n];
 for(int i=0;i<n;i++)
 {
 printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
 scanf("%d %d",&b[i],&p[i]);
 index[i]=i+1;
 }
 for(int i=0;i<n;i++)
 {
 int a=p[i],m=i;
 for(int j=i;j<n;j++)
 {
 if(p[j] > a)
 {
 a=p[j];
 m=j;
 }
 }
 swap(&p[i], &p[m]);
 swap(&b[i], &b[m]);
 swap(&index[i],&index[m]);
 }
 int t=0;
 printf("Order of process Execution is\n");
 for(int i=0;i<n;i++)
```

```
{
printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);
t+=b[i];
}
printf("\n");
printf("Process Id Burst Time Wait Time TurnAround Time\n");
int wait_time=0;
for(int i=0;i<n;i++)
{
printf("P%d %d %d %d\n",index[i],b[i],wait_time,wait_time + b[i]);
wait_time += b[i];
}
return 0;
}
```

Output

```
Enter Number of Processes: 5
Enter Burst Time and Priority Value for Process 1: 4

6
Enter Burst Time and Priority Value for Process 2: 4
7
Enter Burst Time and Priority Value for Process 3: 3
7
Enter Burst Time and Priority Value for Process 4: 2
7
Enter Burst Time and Priority Value for Process 5: 3
8
Order of process Execution is
P5 is executed from 0 to 3
P2 is executed from 3 to 7
P3 is executed from 7 to 10
P4 is executed from 10 to 12
P1 is executed from 12 to 16

Process Id     Burst Time   Wait Time    TurnAround Time
P5          3          0          3
P2          4          3          7
P3          3          7          10
P4          2          10          12
P1          4          12          16

Process returned 0 (0x0)   execution time : 18.056 s
Press any key to continue.
```

# PRACTICAL – 6

**AIM:** Perform Preemptive Priority Scheduling Algorithm.

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

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

int main() {
    int n;
    cout << "Enter Number of Processes: ";
    cin >> n;

    int b[n], p[n], index[n];

    for(int i = 0; i < n; i++) {
        cout << "Enter Burst Time and Priority Value for Process " << i + 1 << ": ";
        cin >> b[i] >> p[i];
        index[i] = i + 1;
    }

    for(int i = 0; i < n; i++) {
        int a = p[i], m = i;
        for(int j = i; j < n; j++) {
            if(p[j] > a) {
                a = p[j];
                m = j;
            }
        }
        swap(&p[i], &p[m]);
        swap(&b[i], &b[m]);
        swap(&index[i], &index[m]);
    }
```

```
    int t = 0;
    cout << "Order of process Execution is" << endl;
    for(int i = 0; i < n; i++) {
        cout << "P" << index[i] << " is executed from " << t << " to " << t + b[i] << endl;
        t += b[i];
    }

    cout << "\nProcess Id Burst Time Wait Time TurnAround Time" << endl;
    int wait_time = 0;
    for(int i = 0; i < n; i++) {
        cout << "P" << index[i] << " " << b[i] << " " << wait_time << " " << wait_time +
b[i] << endl;
        wait_time += b[i];
    }

    return 0;
}
```

Output

```
Preemptive Priority Scheduling Menu:
1. Enter Processes
2. Run Preemptive Priority Scheduling
3. Exit
Enter your choice: 1
Enter the number of processes: 3
Enter burst time for Process 1: 4
Enter priority for Process 1: 1
Enter burst time for Process 2: 6
Enter priority for Process 2: 4
Enter burst time for Process 3: 6
Enter priority for Process 3: 3

Preemptive Priority Scheduling Menu:
1. Enter Processes
2. Run Preemptive Priority Scheduling
3. Exit
Enter your choice: 2

Average Waiting Time: 4.666666666666667
Average Turnaround Time: 10.0

Process ID  Burst Time  Priority    Waiting Time    Turnaround Time
1               4               1               0               4
2               6               4               10              16
3               6               3               4               10
```

# PRACTICAL – 7

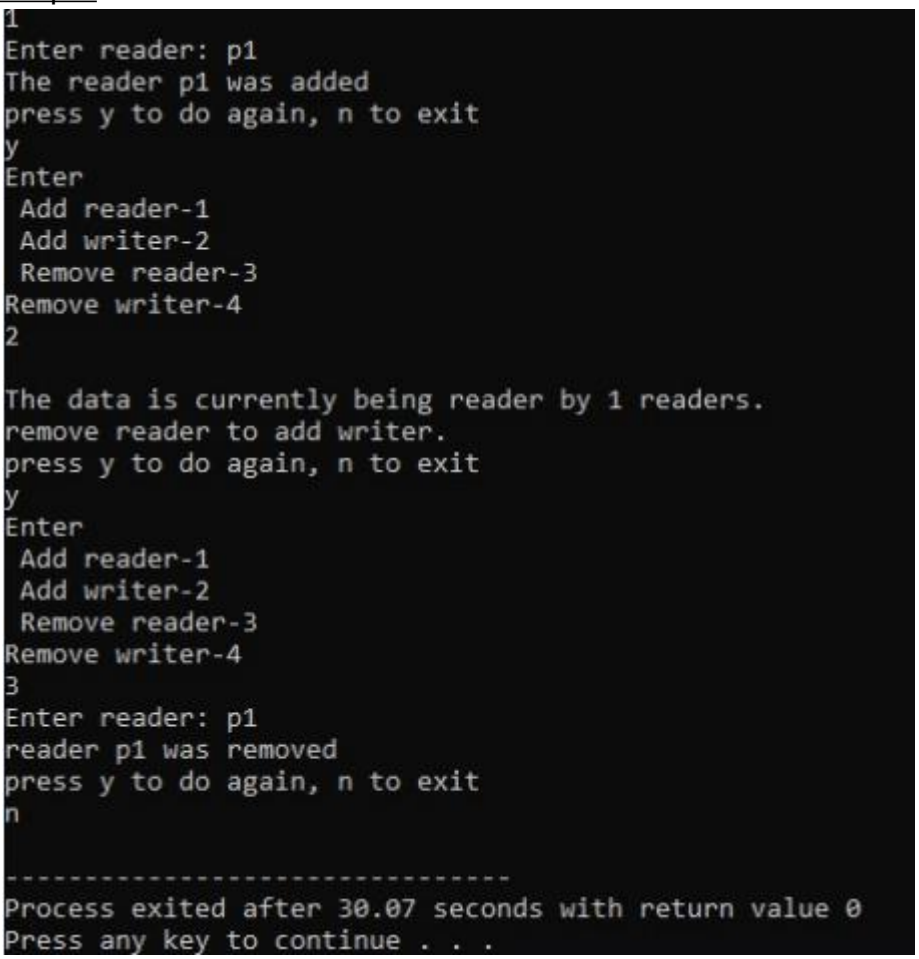
**AIM:** Perform Reader Writer Synchronisation Algorithm.

```
#include<bits/stdc++.h>
using namespace std;
int main(){
int choice;
set<string> reader;
set<string> writer;
char ch='y';
while(ch=='y')
{
cout<<"Enter "<<endl<<" Add reader-1 \n"<<" Add writer-2 \n"<<" Remove reader-
3\n";
cout<<"Remove writer-4\n";
cin>>choice;
if(choice==3)
{
if(reader.size()==0)
{
cout<<"\nThere are no readers\n";
}
else
{
cout<<"Enter reader: ";
string s;
cin>>s;
reader.erase(s);
cout<<"reader "<<s<<" was removed \n";
}
}
else if(choice==1)
{
if(writer.size()!=0){
cout<<"\nCan't add reader as the data is being written by a writer.\n";
}
else{
cout<<"Enter reader: ";
string s;
```

```
cin>>s;
reader.insert(s);
cout<<"The reader "<<s<<" was added \n";
}
}
else if(choice==4)
{
if(writer.size()==0)
{
cout<<"\nThere are no writers\n";
}
else
{
cout<<"Enter writer: ";
string s;
cin>>s;
writer.erase(s);
cout<<"The writer "<<s<<" was removed \n";
}
}
else
{
if(writer.size()!=0){
cout<<"\nCan't add writer as the data is being written by a writer.\n";
}
else if(reader.size()!=0){
cout<<"\nThe data is currently being reader by "<<reader.size()<<" readers.\n";
cout<<"remove reader to add writer.\n";
}
else{
cout<<"Enter name of writer: ";
string s;
cin>>s;
writer.insert(s);
cout<<"The writer "<<s<<" was added \n";
}
}
cout<<"press y to do again, n to exit\n";
cin>>ch;
}
return 0;
```

}

Output


1
Enter reader: p1
The reader p1 was added
press y to do again, n to exit
y
Enter
Add reader-1
Add writer-2
Remove reader-3
Remove writer-4
2
The data is currently being reader by 1 readers.
remove reader to add writer.
press y to do again, n to exit
y
Enter
Add reader-1
Add writer-2
Remove reader-3
Remove writer-4
3
Enter reader: p1
reader p1 was removed
press y to do again, n to exit
n
--------------------------------
Process exited after 30.07 seconds with return value 0
Press any key to continue . . .

# PRACTICAL - 8

**AIM:** Perform Dining Philosophers Synchronization Algorithm

```
#include<stdio.h>
#define n 5
int compltedPhilo = 0,i;
struct fork{
int taken;
}ForkAvil[n];
struct philosp{
int left;
int right;
}Philostatus[n];
void goForDinner(int philID){
if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
printf("Philosopher %d completed his dinner\n",philID+1);
else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){
printf("Philosopher %d completed his dinner\n",philID+1);
Philostatus[philID].left = Philostatus[philID].right = 10;
int otherFork = philID-1;
if(otherFork== -1)
otherFork=(n-1);
ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;
printf("Philosopher %d released fork %d and
fork %d\n",philID+1,philID+1,otherFork+1);
compltedPhilo++;
}
else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){
if(philID==(n-1)){
if(ForkAvil[philID].taken==0){
ForkAvil[philID].taken = Philostatus[philID].right = 1;
printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);
}else{
printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);
}
}else{
int dupphilID = philID;
philID-=1;
if(philID== -1)
philID=(n-1);
if(ForkAvil[philID].taken == 0){
ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
```

```
printf("Fork %d taken by Philosopher %d\n",phiIID+1,dupphiIID+1);
}else{
printf("Philosopher %d is waiting for Fork %d\n",dupphiIID+1,phiIID+1);
}
}
}
else if(Philostatus[phiIID].left==0){
if(phiIID==(n-1)){
if(ForkAvil[phiIID-1].taken==0){
ForkAvil[phiIID-1].taken = Philostatus[phiIID].left = 1;
printf("Fork %d taken by philosopher %d\n",phiIID,phiIID+1);
}else{
printf("Philosopher %d is waiting for fork %d\n",phiIID+1,phiIID);
}
}else{
if(ForkAvil[phiIID].taken == 0){
ForkAvil[phiIID].taken = Philostatus[phiIID].left = 1;
printf("Fork %d taken by Philosopher %d\n",phiIID+1,phiIID+1);
}else{
printf("Philosopher %d is waiting for Fork %d\n",phiIID+1,phiIID+1);
}
}
}else{}
}
int main(){
for(i=0;i<n;i++)
ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;
while(compltedPhilo<n){
for(i=0;i<n;i++)
goForDinner(i);
printf("\nTill now num of philosophers completed dinner are
%d\n\n",compltedPhilo);
}
return 0;
}
```

Output

```
1
Enter reader: p1
The reader p1 was added
press y to do again, n to exit
y
Enter
 Add reader-1
 Add writer-2
 Remove reader-3
Remove writer-4
2

The data is currently being reader by 1 readers.
remove reader to add writer.
press y to do again, n to exit
y
Enter
 Add reader-1
 Add writer-2
 Remove reader-3
Remove writer-4
3
Enter reader: p1
reader p1 was removed
press y to do again, n to exit
n

--------------------------------
Process exited after 30.07 seconds with return value 0
Press any key to continue . . .
```

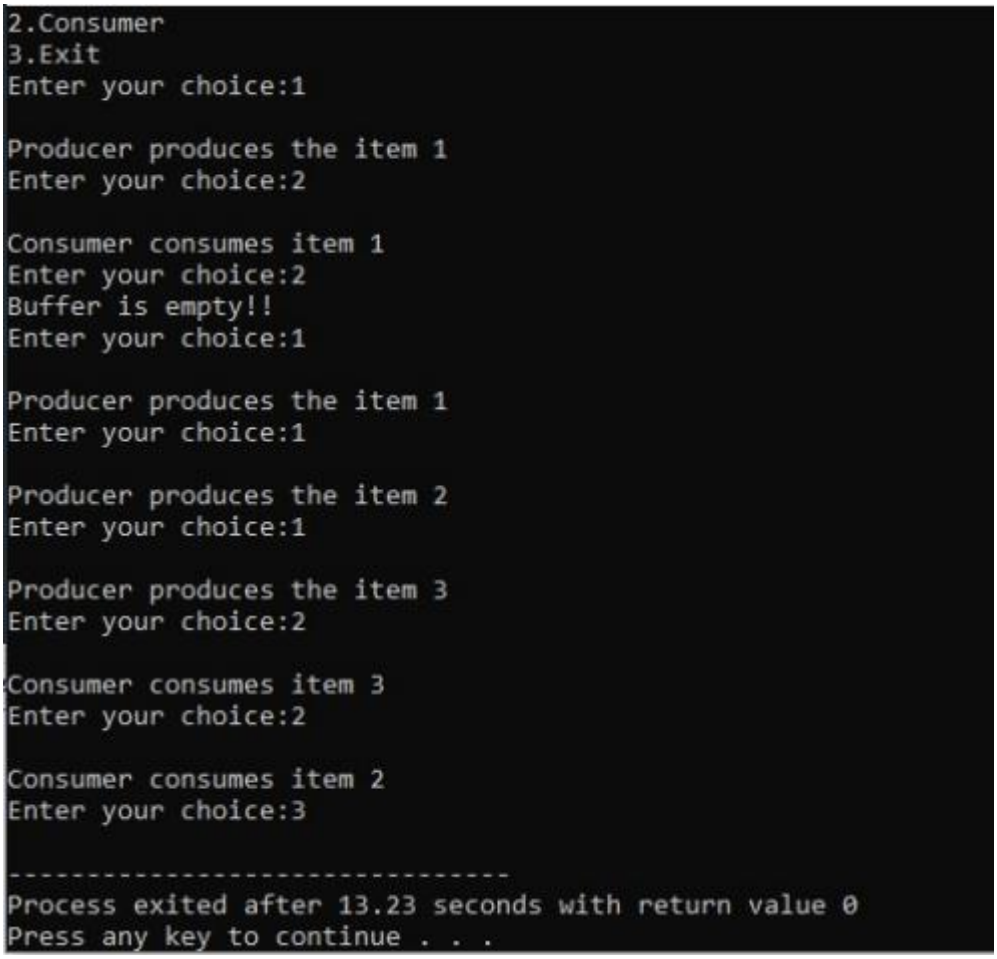
# PRACTICAL - 9

**AIM:** Perform bounded buffer Synchronization Algorithm

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");
while(1)
{
printf("\nEnter your choice:");
scanf("%d",&n);
switch(n)
{
case 1:if((mutex==1)&&(empty!=0))
producer();
else
break;
printf("Buffer is full!!");
 case 2:if((mutex==1)&&(full!=0))
consumer();
case 3:
}
}
else
break;
exit(0);
break;
printf("Buffer is empty!!");
return 0;
}
int wait(int s)
{
return (--s);
}
int signal(int s)
```

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

Output



# PRACTICAL - 10

**AIM:** Perform Banker's Algorithm

```
#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

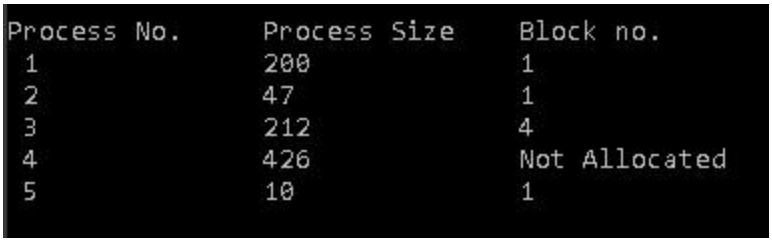
# PRACTICAL - 11

**AIM:** Perform First Fit, Best Fit, and Worst Fit Memory Management Algorithms

First Fit

```
include<bits/stdc++.h>
using namespace std;
void First_Fit(int block_size[], int total_blocks, int process_size[], int total_process) {
  int allocation[total_process];
  memset(allocation, -1, sizeof(allocation));
  for (int i = 0; i < total_process; i++) {
    for (int j = 0; j < total_blocks; j++) {
      if (block_size[j] >= process_size[i]) {
        allocation[i] = j;
        block_size[j] -= process_size[i];
        break;
      }
    }
  }
  cout << "\nProcess No.\tProcess Size\tBlock no.\n";
  for (int i = 0; i < total_process; i++) {
    cout << " " << i+1 << "\t\t" << process_size[i] << "\t\t";
    if (allocation[i] != -1)
      cout << allocation[i] + 1;
    else
      cout << "Not Allocated";
      cout << endl;
  }
}
int main() {
  int block_size[] = {300, 50, 200, 350, 70};
  int process_size[] = {200, 47, 212, 426, 10};
  int total_blocks = sizeof(block_size) / sizeof(block_size[0]);
  int total_process = sizeof(process_size) / sizeof(process_size[0]);
  First_Fit(block_size, total_blocks, process_size, total_process);
  return 0 ;
}
```

Output

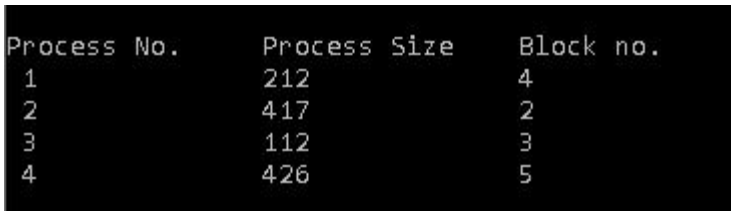


Best Fit

```
#include<iostream>
using namespace std;
void bestFit(int blockSize[], int m, int processSize[], int n)
{
int allocation[n];
for (int i = 0; i < n; i++)
allocation[i] = -1;
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 ;
}
```

Output

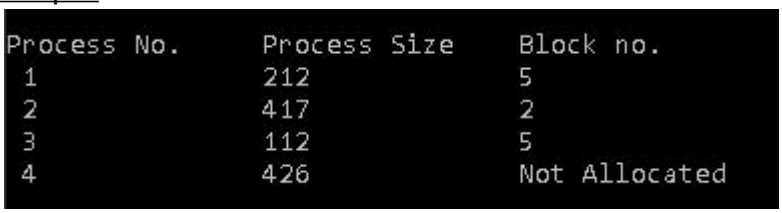



Worst Fit

```
#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



# PRACTICAL – 12

**AIM:** Perform First In First Out Page Replacement Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int getPageFaults(int pages[], int n, int frames)
{
unordered_set <int> set;
queue <int> indexes;
int countPageFaults = 0;
for (int i=0; i < n; i++)
{
if (set.size() < frames)
{
if (set.find(pages[i])==set.end())
{
set.insert(pages[i]);
countPageFaults++;
indexes.push(pages[i]);}}
else
{
if (set.find(pages[i]) == set.end())
{
int val = indexes.front();
indexes.pop();
set.erase(val);
set.insert(pages[i]);
indexes.push(pages[i]);
countPageFaults++;}}}
return countPageFaults;
}
int main()
{
int pages[] = {4, 1, 2, 4, 5};
int n = sizeof(pages)/sizeof(pages[0]);
int frames = 4;
cout << "Page Faults: " << getPageFaults(pages, n, frames);
return 0;
}
```

Output

```
Page Faults: 4
```

# PRACTICAL – 13

**AIM:** Perform Least Recently Used Page Replacement Algorithm

```
#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 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

```
Page Faults: 6
```

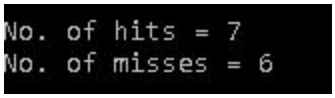
# PRACTICAL – 14

**AIM:** Perform Optimal Page Replacement Page Replacement Algorithm

```
#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 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



# PRACTICAL – 15

**AIM:** Perform First Come First Disk Scheduling Algorithm.

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;
void FCFS(int arr[], int head)
{
int seek_count = 0;
int distance, cur_track;
for (int i = 0; i < size; i++) {
cur_track = arr[i];
distance = abs(cur_track - head);
seek_count += distance;
head = cur_track;
}
cout << "Total number of seek operations = "<< seek_count << endl;
cout << "Seek Sequence is" << endl;
for (int i = 0; i < size; i++) {
cout << arr[i] << endl;
}
}
int main()
{
int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };
int head = 50;
FCFS(arr, head);
return 0;
}
```

Output

```
Total number of seek operations = 510
Seek Sequence is
176
79
34
60
92
11
41
114
```

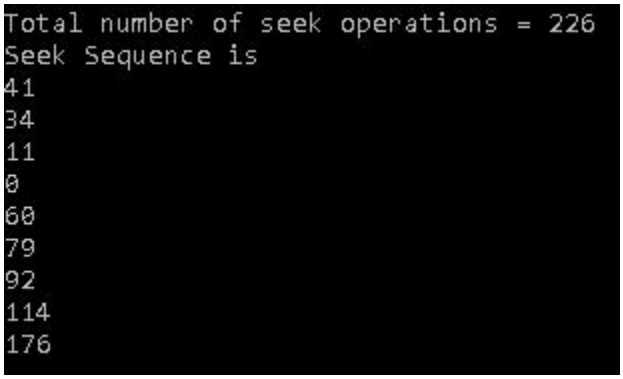
# PRACTICAL – 16

**AIM:** Perform SCAN Disk Scheduling Algorithm.

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;
int disk_size = 200;
void SCAN(int arr[], int head, string direction)
{
int seek_count = 0;
int distance, cur_track;
vector<int> left, right;
vector<int> seek_sequence;
if (direction == "left")
left.push_back(0);
else if (direction == "right")
right.push_back(disk_size - 1);
for (int i = 0; i < size; i++) {
if (arr[i] < head)
left.push_back(arr[i]);
if (arr[i] > head)
right.push_back(arr[i]);
}
std::sort(left.begin(), left.end());
std::sort(right.begin(), right.end());
int run = 2;
while (run--) {
if (direction == "left") {
for (int i = left.size() - 1; i >= 0; i--) {
cur_track = left[i];
seek_sequence.push_back(cur_track);
distance = abs(cur_track - head);
seek_count += distance;
head = cur_track;
}
direction = "right";
}
else if (direction == "right") {
for (int i = 0; i < right.size(); i++) {
cur_track = right[i];
seek_sequence.push_back(cur_track);
distance = abs(cur_track - head);
```

```
seek_count += distance;
head = cur_track;
}
direction = "left";
}
}
cout << "Total number of seek operations = "<< seek_count << endl;
cout << "Seek Sequence is" << endl;
for (int i = 0; i < seek_sequence.size(); i++) {
cout << seek_sequence[i] << endl;
}
}
int main()
{
int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };
int head = 50;
string direction = "left";
SCAN(arr, head, direction);
return 0;
}
```

Output



# PRACTICAL – 17

**AIM:** Perform C-SCAN Disk Scheduling Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;
int disk_size = 200;
void CSCAN(int arr[], int head)
{
int seek_count = 0;
int distance, cur_track;
vector<int> left, right;
vector<int> seek_sequence;
left.push_back(0);
right.push_back(disk_size - 1);
for (int i = 0; i < size; i++) {
if (arr[i] < head)
left.push_back(arr[i]);
if (arr[i] > head)
right.push_back(arr[i]);
}
std::sort(left.begin(), left.end());
std::sort(right.begin(), right.end());
for (int i = 0; i < right.size(); i++) {
cur_track = right[i];
seek_sequence.push_back(cur_track);
distance = abs(cur_track - head);
seek_count += distance;
head = cur_track;
}
head = 0;
seek_count += (disk_size - 1);
for (int i = 0; i < left.size(); i++) {
cur_track = left[i];
seek_sequence.push_back(cur_track);
distance = abs(cur_track - head);
seek_count += distance;
head = cur_track;
}
cout << "Total number of seek operations = "<< seek_count << endl;
cout << "Seek Sequence is" << endl;
for (int i = 0; i < seek_sequence.size(); i++) {
```

```
cout << seek_sequence[i] << endl;
}
}
int main()
{
int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };
int head = 50;
cout << "Initial position of head: " << head << endl;
CSCAN(arr, head);
return 0;
}
```

Output