ASSIGNMENT -2

AIM: Try to process the First Come First Serve (FCFS) Scheduling Algorithm

Technology Stack: C++

Theory:

• **First-Come-First-Serve (FCFS):**

A basic scheduling algorithm for task execution.

• **Order of Arrival:**

Processes are executed in the order they arrive in the ready queue.

• **Non-preemptive:**

Once a process starts execution, it continues until completion.

• **Simple Implementation:**

Easy to understand and implement, making it suitable for simple systems.

• **Fairness Concerns:**

May lead to a phenomenon known as the "convoy effect," where short processes get delayed by long processes.

• **Lack of Optimization:**

Doesn't prioritize task importance or execution time, potentially leading to inefficient resource utilization.

• **Common in Batch Systems:**

Historically used in early computer systems for simplicity in batch processing.

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. forprocess 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_around_time / no_of_processes.

PROGRAM:

```
        Input description.
        First line contains an integer n
        example:
        2
        at    bt
        0     4
        3     6
        */
    #include <bits/stdc++.h>
    using namespace std;
    class Process {
    private:
        int at;
        int bt;
        int ct;
        int tat;
        int wt;
        int pid;
    public:
        int& operator[](string var){
                if (var == "at")
                        return at;
                if (var == "bt")
                        return bt;
                if (var == "ct")
                        return ct;
                if (var == "tat")
                        return tat;
```

```
            if (var == "wt")
                return wt;
            return pid;
    }
    void update_after_ct()
    {
            tat = ct - at;
            wt = tat - bt;
    }
    void display()
    {
            printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
                tat, wt);
    }
};
float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
            total += temp[var];
    }
    return (float)total / P.size();
}
float throughput (vector<Process> P, string var,int n){
   return (float)n/P.back()["ct"];
}
float scheduling_length(vector<Process> P, string var1,string var2){
   return P.back()[var1]-P[0][var2];
}
```

```
void printGanttChart(const vector<pair<int, int>>& gantt) {
        printf("\nGantt Chart:\n");
        for (int i = 0; i < gantt.size(); i++) {
            if (i == 0 || gantt[i].first > gantt[i - 1].second) {
                printf("|%d", gantt[i].first);
            }
            printf("|P%d|%d", i + 1, gantt[i].second);
        }
        printf("|\n");
    }
    int main(){
        int n;
        cin >> n;
        int counter = 0;
        vector<Process> P(n);
        for (Process& temp : P) {
                temp["id"] = counter++;
                cin >> temp["at"] >> temp["bt"];
        }
        sort(P.begin(), P.end(),
                [](Process first, Process second) {
                        return first["at"] < second["at"];
                });
        printf("pid\tat\tbt\tct\ttat\twt\n");
        P[0]["ct"] = P[0]["at"] + P[0]["bt"];
        P[0].update_after_ct();
        P[0].display();

        vector<pair<int, int>> gantt;
```

```
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        if (P[i]["at"] < P[i - 1]["ct"]) {
            P[i]["ct"] = P[i - 1]["ct"] + P[i]["bt"];
        }
        else {
            P[i]["ct"] = P[i]["at"] + P[i]["bt"];
        }
        P[i].update_after_ct();
        P[i].display();
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct",n));
    printf("scheduling length : %f\n", scheduling_length(P, "ct","at"));
    printGanttChart(gantt);
    return 0;
}
```

```
/tmp/LspYjrAEFi.o
5
0 4
3 7
5 5
7 3
8 6
pid at  bt  ct  tat wt
0   0   4   4   4   0
1   3   7   11  8   1
2   5   5   16  11  6
3   7   3   19  12  9
4   8   6   25  17  11
Average waiting time : 5.400000
Average turnaround time : 10.400000
Throughput time : 0.200000
scheduling length : 25.000000

Gantt Chart:
|0|P1|4|P2|11|P3|16|P4|19|P5|25|
```

ASSIGNMENT -3

AIM: Implementation of the Shortest job first (SJF) Scheduling Algorithm

Technology Stack: C++

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
class Process {
private:
   int at;
   int bt;
   int ct;
   int tat;
   int wt;
   int pid;
public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      return pid;
   }
```

```
    void update_after_ct() {
        tat = ct - at;
        wt = tat - bt;
    }
    void display(){
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
               tat, wt);
    }};
float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
float scheduling_length(vector<Process> P);
float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}
float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        }
```

```
    }
    return max_ct - min_at;
}

void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}

void SJF(vector<Process>& processes) {
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["bt"] < processes[b]["bt"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }
    int completed = 0;
    while (completed < n) {
```

```
        int shortest_index = -1;
        int shortest_burst = INT_MAX;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && Remaining_time[i] <
shortest_burst && Remaining_time[i] > 0) {
                shortest_index = i;
                shortest_burst = Remaining_time[i];
                Remaining_time[shortest_index]=0;
                current_time+=processes[process_id]["bt"];
                completed++;
                processes[process_id]["ct"] = current_time;
                processes[process_id].update_after_ct();
            }
        }
        if (shortest_index == -1) {
            current_time++;
        }
    }
}

int main(){
    int n;
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }
```

```
SJF(P);
    printf("pid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);

    return 0;
}
```

```
/tmp/pniNL9BnvT.o
4
2 5
4 4
2 6
18 3
pid at  bt  ct  tat wt
0   2   5   7   5   0
1   4   4   11  7   3
2   2   6   17  15  9
3   18  3   21  3   0
Average waiting time : 3.000000
Average turnaround time : 7.500000
Throughput time : 0.210526
Scheduling length : 19.000000

Gantt Chart:
|2|P1|7|P2|11|P3|17|18|P4|21|
```

ASSIGNMENT -4

AIM: Implementation of the Shortest Remaining Time First(STRF) Scheduling Algorithm

Technology Stack: C++

PROGRAM:

```
 #include <bits/stdc++.h>
 using namespace std;
class Process {
private:
   int at;
   int bt;
   int ct;
   int tat;
   int wt;
   int pid;
public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      return pid;
   }
```

```
 void update_after_ct() {
        tat = ct - at;
        wt = tat - bt;
    }
    void display()  {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
              tat, wt);
    }
};
float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
float scheduling_length(vector<Process> P);
float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}
float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];    } }
```

```
    return max_ct - min_at;
}
void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}
void SRTF(vector<Process>& processes) {
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n); // For maintaining the order of processes based on burst time
    iota(Process_order.begin(), Process_order.end(), 0); // Initialize Process_order with 0, 1, 2, ..., n-1
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["bt"] < processes[b]["bt"]; // Sort Process_order based on burst time
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }
    int completed = 0;
    while (completed < n) {
        int shortest_index = -1;
```

```
        int shortest_burst = INT_MAX;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && Remaining_time[i] <
shortest_burst && Remaining_time[i] > 0) {
                shortest_index = i;
                shortest_burst = Remaining_time[i];
            }
        }
        if (shortest_index == -1) {
            current_time++;
            continue;  }
        int process_id = Process_order[shortest_index];
        Remaining_time[shortest_index]--;
        current_time++;
        if (Remaining_time[shortest_index] == 0) {
            completed++;
            processes[process_id]["ct"] = current_time;
            processes[process_id].update_after_ct();
        }
    }
}
int main()
{
    int n;
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
        temp["id"] = counter++;
```

```
        cin >> temp["at"] >> temp["bt"];
    }
    SRTF(P);
    printf("pid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);
    return 0;
}
```

```
/tmp/fcw9SsnF0c.o
4
0 7
2 3
3 5
6 4
pid at  bt  ct  tat wt
0   0   7   19  19  12
1   2   3   5   3   0
2   3   5   14  11  6
3   6   4   10  4   0
Average waiting time : 4.500000
Average turnaround time : 9.250000
Throughput time : 0.210526
Scheduling length : 19.000000

Gantt Chart:
|0|P1|19|P2|5|P3|14|P4|10|
```

ASSIGNMENT -5

AIM: Implementation of the Highest Response Ratio Next (HRRN) Scheduling Algorithm

Technology Stack: C++

PROGRAM:

```
 #include <bits/stdc++.h>
using namespace std;
class Process {
private:
   int at;
   int bt;
   int ct;
   int tat;
   int wt;
   int pid;
   float rr;
public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
     return pid;}
```

```
  void update_after_ct(){
      tat = ct - at;
      wt = tat - bt;
   }
   void display() {
      printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
            tat, wt);
   }
   float response_ratio(int current_time)  {
      return (float)(current_time - at + bt) / bt;
   }};
float average(vector<Process> P, string var){
   int total = 0;
   for (auto temp : P) {
      total += temp[var];
   }
   return (float)total / P.size();
}
float scheduling_length(vector<Process> P);
float throughput(vector<Process> P, string var, int n){
   return (float)n / scheduling_length(P);
}
float scheduling_length(vector<Process> P){
   int max_ct = INT_MIN;
   int min_at = INT_MAX;
   for (auto temp : P) {
      if (temp["ct"] > max_ct) {
         max_ct = temp["ct"]; }
      if (temp["at"] < min_at) {
```

```
            min_at = temp["at"];
        }
    }
    return max_ct - min_at;
}
void printGanttChart(const vector<pair<int, int>>& gantt){
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}
void HRRN(vector<Process>& processes){
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["at"] < processes[b]["at"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];  }
    int completed = 0;
    while (completed < n) {
        int selected_index = -1;
```

```
        float highest_rr = -1.0;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && Remaining_time[i] > 0) {
                float rr = processes[process_id].response_ratio(current_time);
                if (rr > highest_rr) {
                    highest_rr = rr;
                    selected_index = i;
                }
            }
        }
        if (selected_index == -1) {
            current_time++;
        } else {
            int process_id = Process_order[selected_index];
            current_time += Remaining_time[selected_index];
            Remaining_time[selected_index] = 0;
            completed++;
            processes[process_id]["ct"] = current_time;
            processes[process_id].update_after_ct();
        }
    }
}
int main(){
    int n;
    cin >> n;
     int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
```

```
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }
    HRRN(P);
    printf("pid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();}
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({ P[0]["at"], P[0]["ct"] });
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({ P[i]["at"], P[i]["ct"] });
    }
    printGanttChart(gantt);
    return 0;
}
```

```
/tmp/TTg5F7KOsk.o
5
0 3
2 6
4 4
6 5
8 2
pid at  bt  ct  tat wt
0   0   3   3   3   0
1   2   6   9   7   1
2   4   4   13  9   5
3   6   5   20  14  9
4   8   2   15  7   5
Average waiting time : 4.000000
Average turnaround time : 8.000000
Throughput time : 0.250000
Scheduling length : 20.000000

Gantt Chart:
|0|P1|3|P2|9|P3|13|P4|20|P5|15|
```

ASSIGNMENT -6

AIM: Implementation of the Priority Scheduling Algorithm

Technology Stack: C++

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
class Process {
private:
   int at;
   int bt;
   int ct;
   int tat,wt;
   int pid;
   int pr;
public:
   int& operator[](string var){
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      if (var == "pr")
         return pr;
       return pid; }
```

```
    void update_after_ct() {
        tat = ct - at;
        wt = tat - bt;
    }
    void display() {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt,pt, ct, tat, wt);
    }};
float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
float scheduling_length(vector<Process> P);
float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}
float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        } }
  return max_ct - min_at;}
```

```
 void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}
void priorityScheduling(vector<Process>& processes) {
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["pr"] < processes[b]["pr"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }
    int completed = 0;
    while (completed < n) {
        int highest_priority_index = -1;
        int highest_priority = INT_MAX;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && processes[process_id]["pr"] <
highest_priority &&  Remaining_time[i] > 0) {
```

```
            highest_priority_index = i;
            highest_priority = processes[process_id]["pr"];
            Remaining_time[highest_priority_index]=0;
            current_time+=processes[process_id]["bt"];
            completed++;
            processes[process_id]["ct"] = current_time;
            processes[process_id].update_after_ct();
          }
        }
        if (highest_priority_index == -1) {
          current_time++;
        }
    }
}
int main(){
    int n;
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"] >> temp["pr"]; // Add priority input
    }
    priorityScheduling(P);
    printf("pid\tat\tbt\tpr\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
```

```
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);

    return 0;
}
```

```
/tmp/YM3m4vq6Jt.o
4
1 5 6
3 4 4
2 6 7
7 5 0
pid at  bt  pr  ct  tat wt
0   1   5   6   6   5   0
1   3   4   4   10  7   3
2   2   6   7   21  19  13
3   7   5   0   15  8   3
Average waiting time : 4.750000
Average turnaround time : 9.750000
Throughput time : 0.200000
Scheduling length : 20.000000

Gantt Chart:
|1|P1|6|P2|10|P3|21|P4|15|
```

ASSIGNMENT -7

AIM: Implementation of the Round-Robin Scheduling Algorithm

Technology Stack: C++

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
class Process {
private:
   int at;
   int bt;
   int ct;
   int tat;
   int wt;
   int pid;
public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      return pid;
   }
```

```
    void update_after_ct(){
        tat = ct - at;
        wt = tat - bt;
    }
    void display(){
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
              tat, wt);
    } };
float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
float scheduling_length(vector<Process> P);
float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}
float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        } }
```

```
    return max_ct - min_at;
}
void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}
void roundRobin(vector<Process>& processes, int quantum) {
    int current_time = 0;
    int n = processes.size();
    queue<int> ready_queue;
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["at"] < processes[b]["at"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[i]["bt"];
    }
    int completed = 0;
    int count = 0;
    while (completed < n) {
        for (int i = 0; i < n; ++i) {
```

```
    int process_id = Process_order[i];
    if (processes[process_id]["at"] <= current_time && Remaining_time[i] > 0) {
        bool already_in_queue = false;
        for (int j = 0; j < ready_queue.size(); ++j) {
            if (ready_queue.front() == process_id) {
                already_in_queue = true;
            }
            ready_queue.push(ready_queue.front());
            ready_queue.pop();
        }
        if (!already_in_queue) {
            ready_queue.push(process_id);
        }
    }
}
if(count>0){
            int d = ready_queue.front();
            ready_queue.pop();
            if(Remaining_time[d]!=0){
                ready_queue.push(d);
        } }
if (ready_queue.empty()) {
    current_time++;
    continue;
}
int process_ids = ready_queue.front();
    count++;
int remaining = Remaining_time[process_ids];
if (remaining >= quantum) {
```

```
            current_time += quantum;
            Remaining_time[process_ids] -= quantum;
            int a=Remaining_time[process_ids];
            if(a==0){
               completed++;
               processes[process_ids]["ct"] = current_time;
               processes[process_ids].update_after_ct();
            }
        } else {
            current_time += remaining;
            Remaining_time[process_ids] = 0;
            completed++;
            processes[process_ids]["ct"] = current_time;
            processes[process_ids].update_after_ct();
        }
    }
}
int main(){
    int n, quantum;
    cin >> n >> quantum;
    int counter = 0;
    vector<Process> P(n);
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }
    roundRobin(P, quantum);
    printf("pid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
```

```
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);
    return 0;
}
```

Output

```
6 4
0 5
1 6
2 3
3 1
4 5
6 4
pid at  bt  ct  tat wt
0   0   5   17  17  12
1   1   6   23  22  16
2   2   3   11  9   6
3   3   1   12  9   8
4   4   5   24  20  15
5   6   4   21  15  11
Average waiting time : 11.333333
Average turnaround time : 15.333333
Throughput time : 0.250000
Scheduling length : 24.000000

Gantt Chart:
|0|P1|17|P2|23|P3|11|P4|12|P5|24|P6|21|
```

EXPERIMENT-3

Aim :- Write and implement the code in C/C++ for the SRTF (shortest remaining time first) CPU scheduling Algorithm.

Theory :- SRTF stands for “shortest remaining time first”. This is a type of preemptive scheduling algorithm that uses burst time as its basic criteria.It is a variant of the Shortest Job Next (SJN) algorithm, also known as Shortest Job First (SJF), but with preemption.

CODE

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

class Process {
private:
   int at; int bt; int ct; int tat; int wt; int pid;
public:
   int& operator[](string var)
   {
      if (var == "at")return at;
      if (var == "bt") return bt;
      if (var == "ct") return ct;
      if (var == "tat") return tat;
      if (var == "wt") return wt;
      return pid;
   }
   void update_after_ct() {
      tat = ct - at;
      wt = tat - bt;
   }
   void display() {
      printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
            tat, wt);
   }
};

float average(vector<Process> P, string var)
{
   int total = 0;
```

```
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}

float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        }
    }
    return max_ct - min_at;
}

float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}

void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}

void SRTF(vector<Process>& processes) {
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["bt"] < processes[b]["bt"];
    });
```

```
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }
    int completed = 0;
    while (completed < n) {
        int shortest_index = -1;
        int shortest_burst = INT_MAX;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
                    if (processes[process_id]["at"] <= current_time && Remaining_time[i] <
shortest_burst && Remaining_time[i] > 0) {
                shortest_index = i;
                shortest_burst = Remaining_time[i];
            }
        }
        if (shortest_index == -1) {
            current_time++;
            continue;
        }
        int process_id = Process_order[shortest_index];
        Remaining_time[shortest_index]--;
        current_time++;
        if (Remaining_time[shortest_index] == 0) {
            completed++;
            processes[process_id]["ct"] = current_time;
            processes[process_id].update_after_ct();
        }
    }
}

int main()
{
    int n;
    cout<<"Enter the number of processes\n";
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    cout<<"Give arrival time and burst time in space separated values\n";
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }
    SRTF(P);
    printf("pid\tat\tbt\tct\ttat\twt\n");
```

```
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);
    cout<<endl;

    return 0;
}
```

OUTPUT

```
PS C:\2BitBrain\colleghe\output> cd 'c:\2BitBrain\colleghe\output'
PS C:\2BitBrain\colleghe\output> & .\'srtf.exe'
Enter the number of processes
5
Give arrival time and burst time in space separated values
1 4
3 5
0 3
5 2
0 4
pid     at      bt      ct      tat     wt
0       1       4       9       8       4
1       3       5       18      15      10
2       0       3       3       3       0
3       5       2       7       2       0
4       0       4       13      13      9
Average waiting time : 4.600000
Average turnaround time : 8.200000
Throughput time : 0.277778
Scheduling length : 18.000000

Gantt Chart:
|1|P1|9|P2|18|P3|3|5|P4|7|P5|13|
```

EXPERIMENT-4

Aim :- Write and implement the code in C/C++ for the Priority Scheduling CPU scheduling Algorithm.

Theory :- Priority Scheduling is a non-preemptive algorithm where each process is assigned a priority. The process with the highest priority is selected for execution first. If two processes have the same priority, the one that arrives first is selected. Priority can be based on factors such as burst time, arrival time, memory requirements, or any other criteria deemed important for the scheduling decision.

CODE

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

class Process {
private:
   int at;    int bt;    int ct;    int tat;    int wt;    int pid;    int pr;
public:
   int& operator[](string var)
   {
      if (var == "at")          return at;
      if (var == "bt")          return bt;
      if (var == "ct")          return ct;
      if (var == "tat")          return tat;
      if (var == "wt")          return wt;
      if (var == "pr")          return pr;
      return pid;
   }

   void update_after_ct()    {
      tat = ct - at;
      wt = tat - bt;
   }
   void display()    {
      printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct, tat, wt, pr);
   }
};
float average(vector<Process> P, string var)
{
```

```
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}

float scheduling_length(vector<Process> P);

float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}

float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        }
    }
    return max_ct - min_at;
}

void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}

void priorityScheduling(vector<Process>& processes) {
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
```

```
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["pr"] < processes[b]["pr"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }

    int completed = 0;
    while (completed < n) {
        int highest_priority_index = -1;
        int highest_priority = INT_MAX;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
                if (processes[process_id]["at"] <= current_time && processes[process_id]["pr"] <
highest_priority &&  Remaining_time[i] > 0) {
                highest_priority_index = i;
                highest_priority = processes[process_id]["pr"];
                Remaining_time[highest_priority_index]=0;
                current_time+=processes[process_id]["bt"];
                completed++;
                processes[process_id]["ct"] = current_time;
                processes[process_id].update_after_ct();
            }
        }
        if (highest_priority_index == -1) {
            current_time++;
        }
    }
}
int main()
{
    int n;
    cout<<"Enter the number of processes\n";
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
   cout<<"Give arrival time and burst time and priority in space separated values\n";
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"] >> temp["pr"]; // Add priority input
    }

    priorityScheduling(P);
```

```
    printf("pid\tat\tbt\tct\ttat\twt\tpr\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);cout<<endl;
    return 0;
}
```

OUTPUT

```
PS C:\2BitBrain\colleghe\output> & .\'priority_sched.exe'
Enter the number of processes
5
Give arrival time ,burst time and priority in space separated values
1 3 5
2 1 3
5 7 1
2 6 6
1 3 5
pid     at      bt      ct      tat     wt      pr
0       1       3       4       3       0       5
1       2       1       5       3       2       3
2       5       7       12      7       0       1
3       2       6       21      19      13      6
4       1       3       15      14      11      5
Average waiting time : 5.200000
Average turnaround time : 9.200000
Throughput time : 0.250000
Scheduling length : 20.000000

Gantt Chart:
|1|P1|4|P2|5|P3|12|P4|21|P5|15|
```

EXPERIMENT-5

Aim :- Write and implement the code in C/C++ for the Round Robin CPU scheduling Algorithm.

Theory :- "Round Robin" (RR) scheduling is a preemptive algorithm in which each process is assigned a fixed time unit called a "time quantum" or "time slice." Processes are executed in a cyclic manner, where each process is allowed to run for a time quantum, and then it's moved to the back of the queue, ready to run again when its turn comes up.

CODE

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

class Process {
private:
   int at;int bt;int ct;int tat;int wt;int pid;

public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      return pid;
   }

   void update_after_ct()
   {
      tat = ct - at;
      wt = tat - bt;
   }

   void display()
   {
```

```
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
               tat, wt);
    }
};

float average(vector<Process> P, string var)
{
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}

float scheduling_length(vector<Process> P);

float throughput(vector<Process> P, string var, int n){
    return (float)n/scheduling_length(P);
}

float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        }
    }
    return max_ct - min_at;
}

void printGanttChart(const vector<pair<int, int>>& gantt) {
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}
```

```
void roundRobin(vector<Process>& processes, int quantum) {
    int current_time = 0;
    int n = processes.size();
    queue<int> ready_queue;
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["at"] < processes[b]["at"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[i]["bt"];
    }
    int completed = 0;
    int count = 0;
    while (completed < n) {
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && Remaining_time[i] > 0) {
                bool already_in_queue = false;
                for (int j = 0; j < ready_queue.size(); ++j) {
                    if (ready_queue.front() == process_id) {
                        already_in_queue = true;
                    }
                    ready_queue.push(ready_queue.front());
                    ready_queue.pop();
                }
                if (!already_in_queue) {
                    ready_queue.push(process_id);
                }

            }
        }
        if(count>0)
                {
                    int d = ready_queue.front();
                    ready_queue.pop();
                    if(Remaining_time[d]!=0){
                        ready_queue.push(d);
                }
                }
        if (ready_queue.empty()) {
            current_time++;
```

```
            continue;
        }
        int process_ids = ready_queue.front();
            count++;
        int remaining = Remaining_time[process_ids];
        if (remaining >= quantum) {
            current_time += quantum;
            Remaining_time[process_ids] -= quantum;
            int a=Remaining_time[process_ids];
            if(a==0){
                completed++;
                processes[process_ids]["ct"] = current_time;
                processes[process_ids].update_after_ct();
            }

        } else {
            current_time += remaining;
            Remaining_time[process_ids] = 0;
            completed++;
            processes[process_ids]["ct"] = current_time;
            processes[process_ids].update_after_ct();
        }
    }
}

int main()
{
    int n, quantum;
    cout<<"Enter the number of processes and time quantum\n";
    cin >> n >> quantum;
    int counter = 0;
        vector<Process> P(n);cout<<"Give arrival time and burst time in space separated values\n";
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
    }

    roundRobin(P, quantum);

    printf("pid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
```

```
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({P[0]["at"], P[0]["ct"]});
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({P[i]["at"], P[i]["ct"]});
    }
    printGanttChart(gantt);
    return 0;
}
```

OUTPUT

```
PS C:\2BitBrain\colleghe\output> & .\'round_robin.exe'
Enter the number of processes and time quantum
5
2
Give arrival time and burst time in space separated values
2 4
3 2
1 5
0 2
1 4
pid     at      bt      ct      tat     wt
0       2       4       0       0       0
1       3       2       8       5       3
2       1       5       0       0       0
3       0       2       12      12      10
4       1       4       12      11      7
Average waiting time : 4.000000
Average turnaround time : 5.600000
Throughput time : 0.416667
Scheduling length : 12.000000

Gantt Chart:
|2|P1|0|3|P2|8|P3|0|P4|12|P5|12|
```

EXPERIMENT-6

Aim :- Write and implement the code in C/C++ for the Highest Response Ratio Next CPU scheduling Algorithm.

Theory :- HRRN (Highest Response Ratio Next) is a non-preemptive scheduling algorithm that selects the process with the highest response ratio for execution. Response ratio is calculated based on the waiting time and the burst time of the process. A process with a higher response ratio is given priority.

CODE

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

class Process {
private:
   int at;int bt;int ct;int tat;int wt;int pid;float rr;
public:
   int& operator[](string var)
   {
      if (var == "at")
         return at;
      if (var == "bt")
         return bt;
      if (var == "ct")
         return ct;
      if (var == "tat")
         return tat;
      if (var == "wt")
         return wt;
      return pid;

   }
   void update_after_ct(){
      tat = ct - at;
      wt = tat - bt;
   }
   void display(){
      printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid, at, bt, ct,
            tat, wt);
   }
   float response_ratio(int current_time)
```

```
        return (float)(current_time - at + bt) / bt;
};

float average(vector<Process> P, string var){
    int total = 0;
    for (auto temp : P) {
        total += temp[var];
    }
    return (float)total / P.size();
}
float throughput(vector<Process> P, string var, int n){
    return (float)n / scheduling_length(P);
}
float scheduling_length(vector<Process> P){
    int max_ct = INT_MIN;
    int min_at = INT_MAX;
    for (auto temp : P) {
        if (temp["ct"] > max_ct) {
            max_ct = temp["ct"];
        }
        if (temp["at"] < min_at) {
            min_at = temp["at"];
        }
    }
    return max_ct - min_at;
}
void printGanttChart(const vector<pair<int, int>>& gantt){
    printf("\nGantt Chart:\n");
    for (int i = 0; i < gantt.size(); i++) {
        if (i == 0 || gantt[i].first > gantt[i - 1].second) {
            printf("|%d", gantt[i].first);
        }
        printf("|P%d|%d", i + 1, gantt[i].second);
    }
    printf("|\n");
}

void HRRN(vector<Process>& processes){
    int current_time = 0;
    int n = processes.size();
    vector<int> Remaining_time(n);
    vector<int> Process_order(n);
    iota(Process_order.begin(), Process_order.end(), 0);
```

```
    // Sort processes based on arrival time
    sort(Process_order.begin(), Process_order.end(), [&](int a, int b) {
        return processes[a]["at"] < processes[b]["at"];
    });
    for (int i = 0; i < n; ++i) {
        Remaining_time[i] = processes[Process_order[i]]["bt"];
    }
    int completed = 0;
    while (completed < n) {
        int selected_index = -1;
        float highest_rr = -1.0;
        for (int i = 0; i < n; ++i) {
            int process_id = Process_order[i];
            if (processes[process_id]["at"] <= current_time && Remaining_time[i] > 0) {
                float rr = processes[process_id].response_ratio(current_time);
                if (rr > highest_rr) {
                    highest_rr = rr;
                    selected_index = i;
                }
            }
        }
        if (selected_index == -1) {
            current_time++;
        } else {
            int process_id = Process_order[selected_index];
            current_time += Remaining_time[selected_index];
            Remaining_time[selected_index] = 0;
            completed++;
            processes[process_id]["ct"] = current_time;
            processes[process_id].update_after_ct();
        }
    }
}

int main(){
    cout<<"Enter the number of processes\n";
    int n;
    cin >> n;
    int counter = 0;
    vector<Process> P(n);
    cout<<"Give arrival time and burst time in space separated values\n";
    for (Process& temp : P) {
        temp["id"] = counter++;
        cin >> temp["at"] >> temp["bt"];
```

```
    }

    HRRN(P);

    printf("\npid\tat\tbt\tct\ttat\twt\n");
    for (int i = 0; i < n; i++) {
        P[i].display();
    }
    printf("Average waiting time : %f\n", average(P, "wt"));
    printf("Average turnaround time : %f\n", average(P, "tat"));
    printf("Throughput time : %f\n", throughput(P, "ct", n));
    printf("Scheduling length : %f\n", scheduling_length(P));
    vector<pair<int, int>> gantt;
    gantt.push_back({ P[0]["at"], P[0]["ct"] });
    for (int i = 1; i < P.size(); i++) {
        gantt.push_back({ P[i]["at"], P[i]["ct"] });
    }
    printGanttChart(gantt);cout<<endl;
    return 0;
}
```

OUTPUT

```
PS C:\2BitBrain\colleghe\output> & .\'hrrn.exe'
Enter the number of processes
5
Give arrival time and burst time in space separated values
1 4
0 4
2 6
1 3
6 5
pid     at      bt      ct      tat     wt
0       1       4       11      10      6
1       0       4       4       4       0
2       2       6       17      15      9
3       1       3       7       6       3
4       6       5       22      16      11
Average waiting time : 5.800000
Average turnaround time : 10.200000
Throughput time : 0.227273
Scheduling length : 22.000000

Gantt Chart:
|1|P1|11|P2|4|P3|17|P4|7|P5|22|
PS C:\2BitBrain\colleghe\output>
```

EXPERIMENT NO .8
BANKER'S ALGORITHM

OBJECTIVE
Write a C/C++ program to implement banker's algorithm.

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

class Process{
public:
   int id;
   int num_of_resources;
   int *resources_assigned,*resources_max_need;
   Process(int id, int a, int b, int c, int d) : id(id){
      resources_assigned = new int[2];
      resources_max_need = new int[2];
      resources_assigned[0] = a;
      resources_assigned[1] = b;
      resources_max_need[0] = c;
      resources_max_need[1] = d;
   }
   Process(int id, int num_of_resources) : id(id), num_of_resources(num_of_resources){
      resources_assigned = new int[num_of_resources];
      resources_max_need = new int[num_of_resources];
      for (int i = 0; i < num_of_resources; I++){
         cout << "Enter assigned R" << i << " for process P" << id << " : ";
         cin >> resources_assigned[i];
      }
      for (int i = 0; i < num_of_resources; I++){
         cout << "Enter max needed R" << i << " for process P" << id << " : ";
         cin >> resources_max_need[i];
      }
   }
};

bool enough_resources(vector<int> &available_resources, Process process){
   for (int i = 0; i < available_resources.size(); I++){
      if (available_resources[i] + process.resources_assigned[i] < process.resources_max_need[i])
         return false;
   }
   return true;}
void execute_process(Process &process, int to_execute, vector<int> &available_resources, vector<bool>
&executed, int &count_executed, vector<int> &sequence){
   for (int i = 0; i < available_resources.size(); I++){
      available_resources[i] += process.resources_assigned[i];
   }
   executed[to_execute] = true;
   count_executed++;
   sequence.push_back(to_execute);
}
void print_sequence(vector<int> &sequence, vector<int> &available_resources, vector<Process>
&processes){
   for (int i = 0; i < sequence.size(); I++){
      cout << "Available     : ";
      for (int j = 0; j < available_resources.size(); j++){
         cout << available_resources[j] << " ";
      }
```

```
        cout << “\nP" << i << " released   : ";
        for (int j = 0; j < available_resources.size(); j++){
            available_resources[j] += processes[sequence[i]].resources_assigned[j];
            cout << processes[sequence[i]].resources_assigned[j] << " ";
        }
        cout << endl;
    }
    cout << "Available     : ";
    for (int i = 0; i < available_resources.size(); I++){
        cout << available_resources[i] << " ";
    }
    cout << endl  << endl;
}
void print_table(vector<Process> &processes, vector<int> &available)
{
    cout << "\n---------------------------------------------------------------------\n";
    cout << "Process       "
         << "Allocated       "
         << "Max             "
         << "Need          ";
    cout << "\n---------------------------------------------------------------------\n";
    for (int i = 0; i < processes.size(); i++)
    {
        cout << "P" << i << "             ";
        for (int j = 0; j < available.size(); j++){
            cout << processes[i].resources_assigned[j] << " ";
        }
        cout << "            ";
        for (int j = 0; j < available.size(); j++)
        {
            cout << processes[i].resources_max_need[j] << " ";
        }
        cout << "            ";
        for (int j = 0; j < available.size(); j++)
        {
            cout << processes[i].resources_max_need[j] - processes[i].resources_assigned[j] << " ";
        }
        cout << "\n---------------------------------------------------------------------\n";
    }
}
void bankers_algorithm(vector<Process> &processes, int num_of_resources, vector<int>
&available_resources)
{
    int n = processes.size();
    int count_executed = 0;
    vector<bool> executed(n, false);
    vector<int> copy_av_res = available_resources;
    vector<int> sequence;
    while (count_executed < n)
    {
        int to_execute = -1;
        for (int i = 0; i < n; i++)
        {   if (!executed[i] && enough_resources(available_resources, processes[i]))
            {  to_execute = i;
               break;
            }}
        if (to_execute == -1)
        {
            // Condition of Deadlock
            cout << "!!!!!!!! Deadlock !!!!!!!!";
```

```
            return;}
        execute_process(processes[to_execute], to_execute, available_resources, executed, count_executed,
sequence);
    }  print_table(processes, available_resources);
    print_sequence(sequence, copy_av_res, processes);
}
int main()
{// CASE 1 :
    vector<Process> p = {Process(0, 0, 1, 2, 2),
                         Process(1, 1, 0, 2, 1),
                         Process(2, 1, 1, 2, 3)};
    vector<int> avl = {2, 2};
    bankers_algorithm(p, 2, avl);
    // CASE 2 :
    int n;
    cout << "Enter number of processes : ";
    cin >> n;
    int num_of_resources;
    cout << "Enter number of resources : ";
    cin >> num_of_resources;
    vector<Process> processes;
    for (int i = 0; i < n; i++)
    { cout << "Process P" << i << " details : \n";
       Process new_p(i, num_of_resources);
       processes.push_back(new_p); }
    vector<int> available_resources(num_of_resources);
    for (int i = 0; i < num_of_resources; i++)
    {  cout << "Enter number of available instaces for R" << i << " : ";
       cin >> available_resources[i];
    } bankers_algorithm(processes, num_of_resources, available_resources);
    return 0;
}
```

OUTPUT

A.PEDEFINED DATA

```
------------------------------------------------------------------------
Process       Allocated       Max             Need
------------------------------------------------------------------------
P0            0 1             2 2             2 1
------------------------------------------------------------------------
P1            1 0             2 1             1 1
------------------------------------------------------------------------
P2            1 1             2 3             1 2
------------------------------------------------------------------------
Available     : 2 2
P0 released   : 0 1
Available     : 2 3
P1 released   : 1 0
Available     : 3 3
P2 released   : 1 1
Available     : 4 4
```

OUTPUT (USER DEFINED DATA)

```
Enter number of processes : 3
Enter number of resources : 3
Process P0 details :
Enter assigned R0 for process P0 : 0
Enter assigned R1 for process P0 : 1
Enter assigned R2 for process P0 : 0
Enter max needed R0 for process P0 : 2
Enter max needed R1 for process P0 : 4
Enter max needed R2 for process P0 : 1
Process P1 details :
Enter assigned R0 for process P1 : 1
Enter assigned R1 for process P1 : 1
Enter assigned R2 for process P1 : 0
Enter max needed R0 for process P1 : 2
Enter max needed R1 for process P1 : 2
Enter max needed R2 for process P1 : 2
Process P2 details :
Enter assigned R0 for process P2 : 1
Enter assigned R1 for process P2 : 2
Enter assigned R2 for process P2 : 1
Enter max needed R0 for process P2 : 3
Enter max needed R1 for process P2 : 2
Enter max needed R2 for process P2 : 3
Enter number of available instaces for R0 : 4
Enter number of available instaces for R1 : 4
Enter number of available instaces for R2 : 4

---------------------------------------------------------------------
Process       Allocated       Max             Need
---------------------------------------------------------------------
P0            0 1 0             2 4 1             2 3 1
---------------------------------------------------------------------
P1            1 1 0             2 2 2             1 1 2
---------------------------------------------------------------------
P2            1 2 1             3 2 3             2 0 2
---------------------------------------------------------------------
Available     : 4 4 4
P0 released   : 0 1 0
Available     : 4 5 4
P1 released   : 1 1 0
Available     : 5 6 4
P2 released   : 1 2 1
Available     : 6 8 5
```

Peterson

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
int turn = 0;
int flag[2] = {0};
int c = 0, in = 0, out = 0;
int buffer[10];

void lock(int p)
{
   int other = 1 - p;
   flag[p] = 1;
   turn = p;
   while (flag[other] == 1 && turn == p)
      ;
}

void unlock(int p)
{
   flag[p] = 0;
}

void display()
```

```
{
   cout << "The buffer is: ";
   for (int i = 0; i < 10; i++)
   {
      cout << buffer[i] << " ";
   }
   cout << endl;
}

void producer()
{
   lock(0);
   for (int i = 0; i < 5; i++)
   {
      while (c == 10)
         ;
      buffer[in] = 100 + i * 2;
      cout << "Producer produced: " << buffer[in] << endl;
      in = (in + 1) % 10;
      c++;
      display();
   }
   unlock(0);
}

void consumer()
```

```
{
    lock(1);
    for (int i = 0; i < 5; i++)
    {
        while (c == 0)
            ;
        int k = buffer[out];
        cout << "Consumer consumed: " << k << endl;
        buffer[out] = 0;
        out = (out + 1) % 10;
        c--;
        display();
    }
    unlock(1);
}

int main()
{
    thread t1(producer);
    thread t2(consumer);
    t1.join();
    t2.join();
}
```

Banker problem