Operating System (CS301)

Assignment - 6

**U19CS012**

Write a program for the simulation of

Basic Description

* Take “**n**” no. of process from user with **arrival\_time** and **burst\_time**.
* **arrival\_time** and **burst\_time** should be generated randomly and Compute *Completion Time*, *Turnaround (TAT) Time* and *Waiting Time*.
* Also show the Count of Context Switching in all of the algorithms.

1. Shortest Job First (SJF)

* Process which have the **Shortest Burst Time** are scheduled first.
* If two processes have same Burst Time then **FCFS** is used to break the tie.
* It is a **Non-Preemptive** scheduling algorithm.
* In non-preemptive scheduling, once the CPU cycle is allocated to process, the process **holds it** till it reaches a waiting state or terminated.

**Code**

*// Shortest Job First Algorithm - BHAGYA RANA [U19CS012]*

*#include* <bits/stdc++.h>

using namespace std;

*// "Process" Class*

class Process

{

public:

*// Process ID*

    int id;

*// Arrival Time: Time at which the process arrives in the ready queue.*

    int arrivalTime;

*// Burst Time: Time required by a process for CPU execution.*

    int burstTime;

*// Completion Time: Time at which process completes its execution.*

    int completionTime;

*// Turn Around Time: Time Difference between completion time and arrival time.*

    int turnaroundTime;

*// Waiting Time(W.T): Time Difference between turn around time and burst time.*

    int waitingTime;

*// Constructor for Intializing*

    Process(int id, int arrivalTime, int burstTime)

    {

*this*->id = id;

*this*->arrivalTime = arrivalTime;

*this*->burstTime = burstTime;

    }

};

*// Compare F(x) for Sorting p*

bool compare(const Process &p1, const Process &p2);

*// Function for Shortest Job First Scheduling*

void shortestJobFirst(vector<Process> &p);

*// F(x) to Generate Random Input*

vector<Process> randomInputGenerator(int n);

*// F(x) to Print the Output in Well Structured Format*

void printOutput(vector<Process> &p);

*// F(x) to Call the Algorithm*

void solve(vector<Process> &p);

int main()

{

    cout << "--------------------SHORTEST JOB FIRST ALGORITHM--------------------\n";

    int n;

    cout << "Enter Number of Processes : ";

    cin >> n;

    char choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

    cout << "1 -> Random Input\n2 -> User Input\n\n";

    cout << "Enter Your Choice : ";

    cin >> choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

    vector<Process> p;

*if* (choice == '1')

    {

        p = randomInputGenerator(n);

        cout << "Randomly generated inputs are : " << endl;

*for* (int i = 0; i < n; i++)

        {

            cout << "Arrival time and Burst time of Process " << p[i].id << " : " << p[i].arrivalTime << " " << p[i].burstTime << endl;

        }

    }

*else* *if* (choice == '2')

    {

*for* (int i = 0; i < n; i++)

        {

            int arriveTime, burstTime;

            cout << "Enter Arrival time and Burst time of Process " << i + 1 << " : ";

            cin >> arriveTime >> burstTime;

            p.push\_back(Process(i + 1, arriveTime, burstTime));

        }

    }

*else*

    {

        cout << "Incorrect Input Entered\n";

*return* 0;

    }

    solve(p);

*return* 0;

}

*// Customized function for sorting*

bool compare(const Process &p1, const Process &p2)

{

*if* (p1.arrivalTime == p2.arrivalTime)

    {

*return* p1.burstTime < p2.burstTime;

    }

*return* p1.arrivalTime < p2.arrivalTime;

}

*// Function for Shortest Job First Scheduling*

void shortestJobFirst(vector<Process> &p)

{

    int temp, curr;

*// 1st process would complete 1st.*

    p[0].completionTime = p[0].arrivalTime + p[0].burstTime;

    p[0].turnaroundTime = p[0].completionTime - p[0].arrivalTime;

    p[0].waitingTime = p[0].turnaroundTime - p[0].burstTime;

*// Now for other processes...*

*for* (int i = 1; i < p.size(); i++)

    {

        temp = p[i - 1].completionTime;*// completion time*

*// Burst Time: Time required by a process for CPU execution.*

        int minBurst = p[i].burstTime;

*// Current index for swapping.*

        curr = -1;

*while* (curr == -1)

        {

*for* (int j = i; j < p.size(); j++)

            {

*// If completion time >= arrive time =>[Implies] a process is in queue*

*// if minBurst >= burstTime of curr process then, we need to swap.*

*if* (temp >= p[j].arrivalTime and minBurst >= p[j].burstTime)

                {

                    minBurst = p[j].burstTime;

                    curr = j;

                }

            }

*// If no such process found*

*if* (curr == -1)

                temp = p[i].arrivalTime;

        }

*// Time at which process completes its execution.= Completion Time*

        p[curr].completionTime = temp + p[curr].burstTime;

*// Turn Around Time = Completion Time – Arrival Time*

        p[curr].turnaroundTime = p[curr].completionTime - p[curr].arrivalTime;

*// Waiting Time = Turn Around Time – Burst Time*

        p[curr].waitingTime = p[curr].turnaroundTime - p[curr].burstTime;

*// If process is found then swapping it with ith process.*

        Process temp = p[i];

        p[i] = p[curr];

        p[curr] = temp;

    }

}

*// Function to generate random input*

vector<Process> randomInputGenerator(int n)

{

    unsigned seed = chrono::system\_clock::now().time\_since\_epoch().count();

    default\_random\_engine generator(seed);

    uniform\_int\_distribution<int> d1(0, ((10 \* n) + 1) / 2);

    uniform\_int\_distribution<int> d2(1, 10);

    vector<Process> p;

*for* (int i = 0; i < n; i++)

    {

        p.push\_back(Process(i + 1, d1(generator), d2(generator)));

        d1.reset();

    }

*return* p;

}

*// Referrence : https://stackoverflow.com/questions/14765155/how-can-i-easily-format-my-data-table-in-c*

template <typename T>

void printElement(T t)

{

    cout << left << setw(17) << setfill(' ') << t;

}

*// Utility function to print Output*

void printOutput(vector<Process> &p)

{

    int TotalWaiting = 0, TotalTurnAround = 0;

    cout << "\n----------------------------------SCHEDULED---PROCESS---DETAILS----------------------------------\n\n";

    printElement("Process ID");

    printElement("Arrival Time");

    printElement("Burst Time");

    printElement("Completion Time");

    printElement("Turn Around Time");

    printElement("Waiting Time");

    cout << endl;

*for* (int i = 0; i < p.size(); i++)

    {

        cout << "     ";

        printElement(p[i].id);

        printElement(p[i].arrivalTime);

        printElement(p[i].burstTime);

        printElement(p[i].completionTime);

        printElement(p[i].turnaroundTime);

        printElement(p[i].waitingTime);

        cout << endl;

        TotalWaiting += p[i].waitingTime;

        TotalTurnAround += p[i].turnaroundTime;

    }

    cout << "\n----------------------------------SCHEDULED---PROCESS---SUMMARY----------------------------------\n\n";

    cout << "Average Waiting Time : " << (double)TotalWaiting / (double)p.size() << endl;

    cout << "Average Turn Around Time : " << (double)TotalTurnAround / (double)p.size() << endl;

    cout << "Context Switch : 0\n\n";

}

*// Function which will call other needed functions.*

void solve(vector<Process> &p)

{

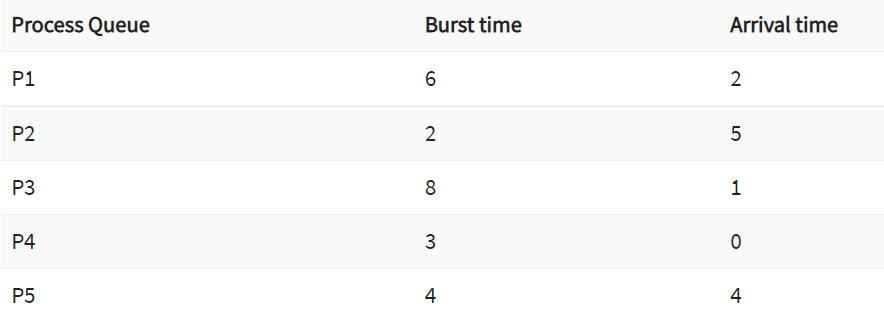
    sort(p.begin(), p.end(), compare);

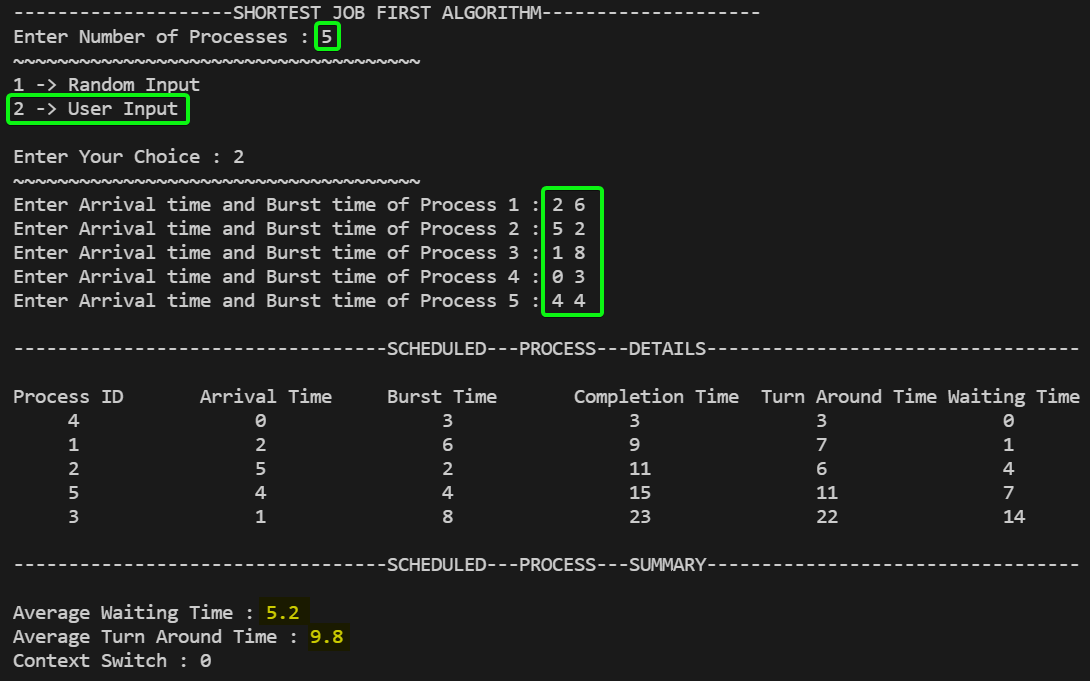
    shortestJobFirst(p);

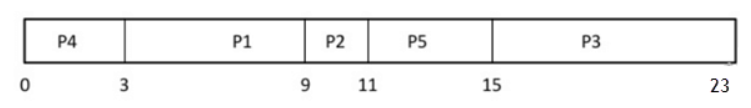
    printOutput(p);

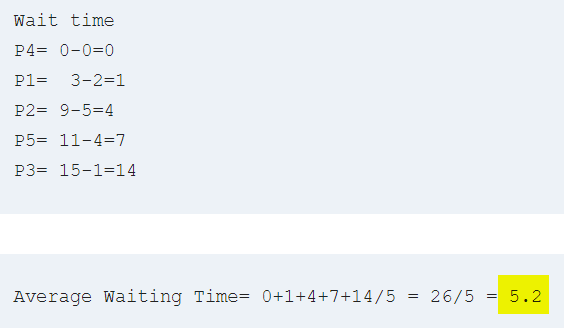
}

**Output** [Non-Preemptive SJF]

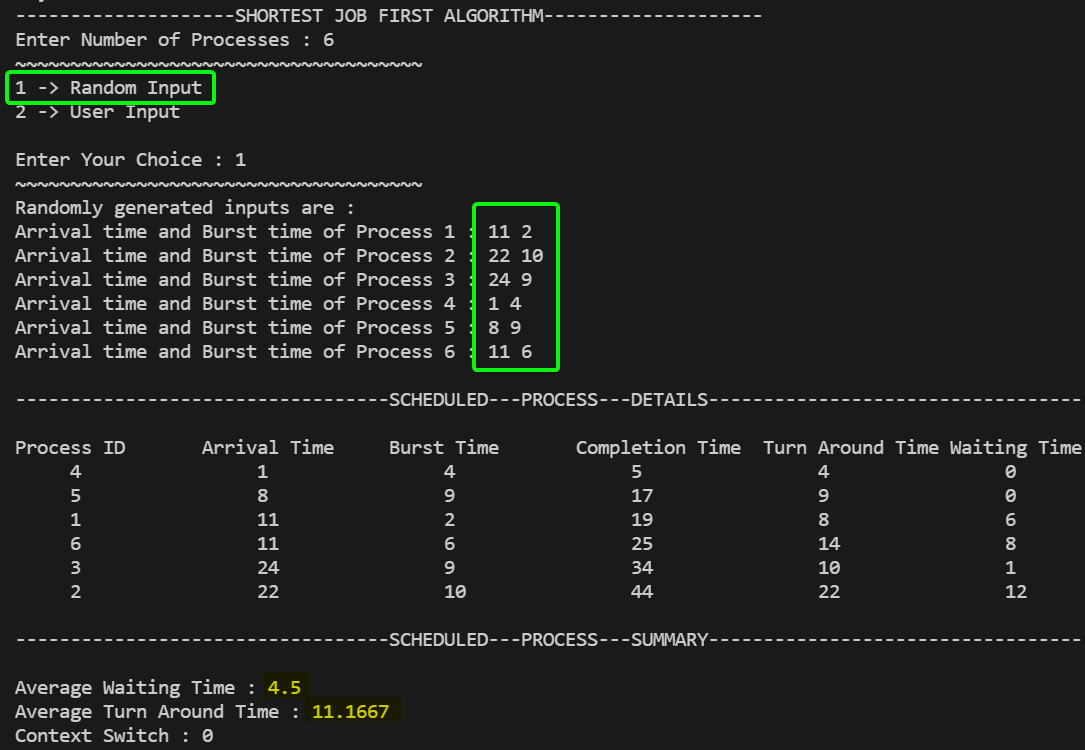








Random Input



2. Shortest Remaining Time First (SRTF) CPU scheduler.

* **Preemptive** mode of **Shortest Job First** is called as *Shortest Remaining Time First (SRTF).*
* In SRTF, the Execution of Process can be stopped after certain amount of time.

**Code**

*// Shortest Remaining Time First Algorithm - BHAGYA RANA [U19CS012]*

*#include* <bits/stdc++.h>

using namespace std;

*// "Process" Class*

class Process

{

public:

*// Process ID*

    int id;

*// Arrival Time: Time at which the process arrives in the ready queue.*

    int arrivalTime;

*// Burst Time: Time required by a process for CPU execution.*

    int burstTime;

*// Completion Time: Time at which process completes its execution.*

    int completionTime;

*// Turn Around Time: Time Difference between completion time and arrival time.*

    int turnaroundTime;

*// Waiting Time(W.T): Time Difference between turn around time and burst time.*

    int waitingTime;

    int remainingTime;

*// Constructor for Intializing*

    Process(int id, int arrivalTime, int burstTime)

    {

*this*->id = id;

*this*->arrivalTime = arrivalTime;

*this*->burstTime = burstTime;

*this*->remainingTime = burstTime;

    }

};

*// Compare F(x) for Sorting p*

bool compare(const Process &p1, const Process &p2);

*// Function for Shortest Remaining Time First Scheduling*

void shortestRemainingTimeFirst(vector<Process> &p);

*// F(x) to Generate Random Input*

vector<Process> randomInputGenerator(int n);

*// F(x) to Print the Output in Well Structured Format*

void printOutput(vector<Process> &p);

*// F(x) to Call the Algorithm*

void solve(vector<Process> &p);

int main()

{

    cout << "--------------------SHORTEST REMAINING TIME FIRST ALGORITHM--------------------\n";

    int n;

    cout << "Enter Number of Processes : ";

    cin >> n;

    char choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

    cout << "1 -> Random Input\n2 -> User Input\n\n";

    cout << "Enter Your Choice : ";

    cin >> choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n";

    vector<Process> p;

*if* (choice == '1')

    {

        p = randomInputGenerator(n);

        cout << "Randomly generated inputs are : " << endl;

*for* (int i = 0; i < n; i++)

        {

            cout << "Arrival time and Burst time of Process " << p[i].id << " : " << p[i].arrivalTime << " " << p[i].burstTime << endl;

        }

    }

*else* *if* (choice == '2')

    {

*for* (int i = 0; i < n; i++)

        {

            int arriveTime, burstTime;

            cout << "Enter Arrival time and Burst time of Process " << i + 1 << " : ";

            cin >> arriveTime >> burstTime;

            p.push\_back(Process(i + 1, arriveTime, burstTime));

        }

    }

*else*

    {

        cout << "Incorrect Input Entered\n";

*return* 0;

    }

    solve(p);

*return* 0;

}

*// Customized function for sorting*

bool compare(const Process &p1, const Process &p2)

{

*if* (p1.arrivalTime == p2.arrivalTime)

*return* p1.burstTime < p2.burstTime;

*return* p1.arrivalTime < p2.arrivalTime;

}

*// Function for Shortest Job First Scheduling*

void shortestRemainingTimeFirst(vector<Process> &p)

{

*// Count No. of Context Switch*

    int contextSwitch = 0, done = 0, currTime = 0, minRemainTime = INT\_MAX, minProcess = 0;

    bool flag = false;

*// For Gantt Chart*

    string time, processFlow;

*// Loop till all processes are completed.*

*while* (done != p.size())

    {

        time += to\_string(currTime) + " ";

*if* (currTime < 10)

            time += " ";

        bool flagTemp = false;

*//Checking every process to find process with minimum remianing time.*

*for* (int i = 0; i < p.size(); i++)

        {

*// Condition for minimum remaining time.*

*if* (p[i].arrivalTime <= currTime and p[i].remainingTime < minRemainTime and p[i].remainingTime > 0)

            {

*// a new process is executed i.e. no context switch*

*if* (minRemainTime == INT\_MAX)

                    flagTemp = true;

                minRemainTime = p[i].remainingTime;

                minProcess = i;

                flag = true;

*if* (flagTemp)

*continue*;

*// Increasing context switch count*

                contextSwitch += 1;

            }

        }

*// No process in queue with minimum remaining time.*

*if* (!flag)

        {

*if* (minRemainTime == INT\_MAX)

                processFlow += "    ";

*else*

                processFlow += "P" + to\_string(p[minProcess].id) + " ";

            currTime++;

*continue*;

        }

        processFlow += "P" + to\_string(p[minProcess].id) + " ";

*//Decrementing remaining time of the feasible process.*

        p[minProcess].remainingTime -= 1;

*// Update minRemainTime*

        minRemainTime = p[minProcess].remainingTime;

*if* (minRemainTime == 0)

        {

*// Process is completed therefore resetting.*

            minRemainTime = INT\_MAX;

*// Increase completed process count*

            done += 1;

            flag = false;

*// Updating Data for a process*

            p[minProcess].completionTime = currTime + 1;

            p[minProcess].waitingTime = p[minProcess].completionTime - p[minProcess].burstTime - p[minProcess].arrivalTime;

*if* (p[minProcess].waitingTime < 0)

                p[minProcess].waitingTime = 0;

            p[minProcess].turnaroundTime = p[minProcess].waitingTime + p[minProcess].burstTime;

        }

        currTime += 1;

    }

    cout << "\n------------------------------GANTT---CHART------------------------------\n";

    cout << "Time   : " + time + to\_string(currTime) << endl;

    cout << "Process    : " + processFlow << endl;

    cout << "Total Number of Context Switching : " << contextSwitch << endl;

}

*// Function to generate random input*

vector<Process> randomInputGenerator(int n)

{

    unsigned seed = chrono::system\_clock::now().time\_since\_epoch().count();

    default\_random\_engine generator(seed);

    uniform\_int\_distribution<int> d1(0, ((10 \* n) + 1) / 2);

    uniform\_int\_distribution<int> d2(1, 10);

    vector<Process> p;

*for* (int i = 0; i < n; i++)

    {

        p.push\_back(Process(i + 1, d1(generator), d2(generator)));

        d1.reset();

    }

*return* p;

}

*// Referrence : https://stackoverflow.com/questions/14765155/how-can-i-easily-format-my-data-table-in-c*

template <typename T>

void printElement(T t)

{

    cout << left << setw(17) << setfill(' ') << t;

}

*// Utility function to print Output*

void printOutput(vector<Process> &p)

{

    int TotalWaiting = 0, TotalTurnAround = 0;

    cout << "\n----------------------------------SCHEDULED---PROCESS---DETAILS----------------------------------\n\n";

    printElement("Process ID");

    printElement("Arrival Time");

    printElement("Burst Time");

    printElement("Completion Time");

    printElement("Turn Around Time");

    printElement("Waiting Time");

    cout << endl;

*for* (int i = 0; i < p.size(); i++)

    {

        cout << "     ";

        printElement(p[i].id);

        printElement(p[i].arrivalTime);

        printElement(p[i].burstTime);

        printElement(p[i].completionTime);

        printElement(p[i].turnaroundTime);

        printElement(p[i].waitingTime);

        cout << endl;

        TotalWaiting += p[i].waitingTime;

        TotalTurnAround += p[i].turnaroundTime;

    }

    cout << "\n----------------------------------SCHEDULED---PROCESS---SUMMARY----------------------------------\n\n";

    cout << "Average Waiting Time : " << (double)TotalWaiting / (double)p.size() << endl;

    cout << "Average Turn Around Time : " << (double)TotalTurnAround / (double)p.size() << endl;

}

*// Function which will call other needed functions.*

void solve(vector<Process> &p)

{

    sort(p.begin(), p.end(), compare);

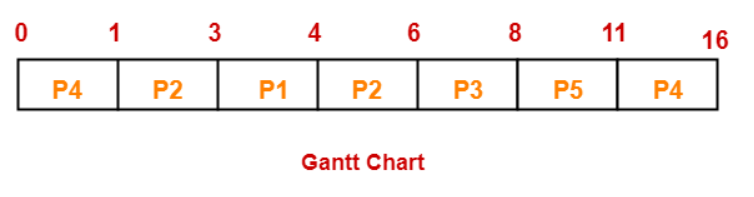
    shortestRemainingTimeFirst(p);

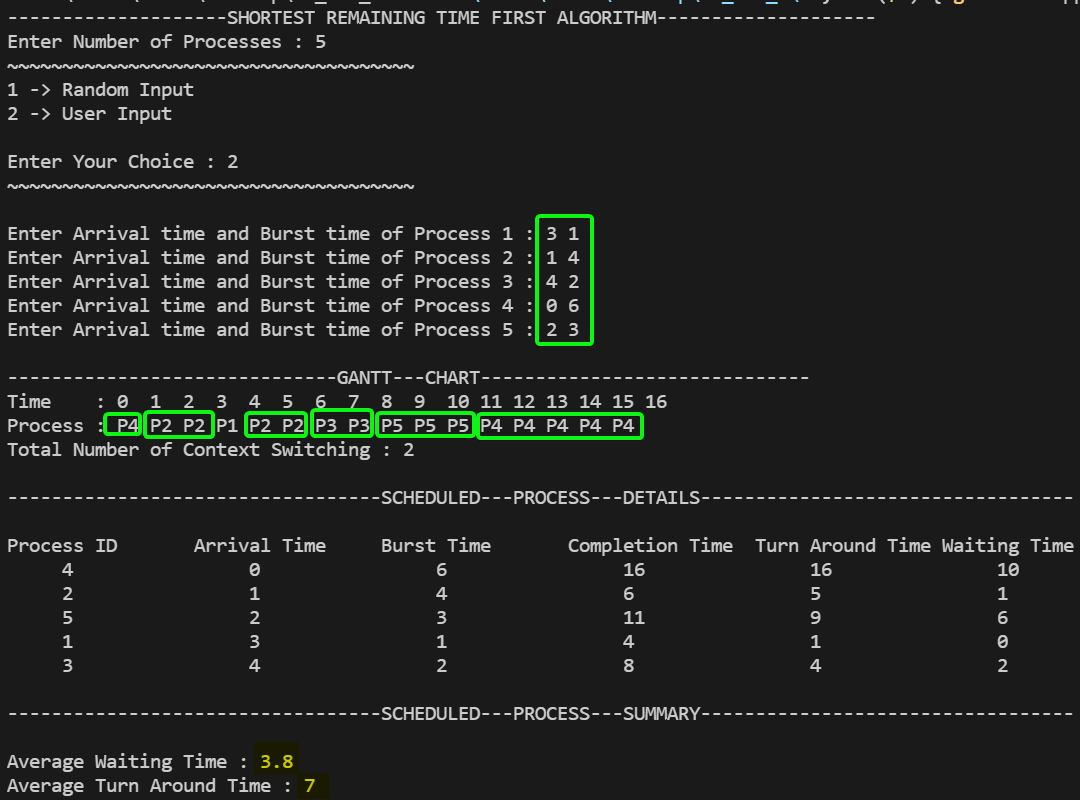
    printOutput(p);

}

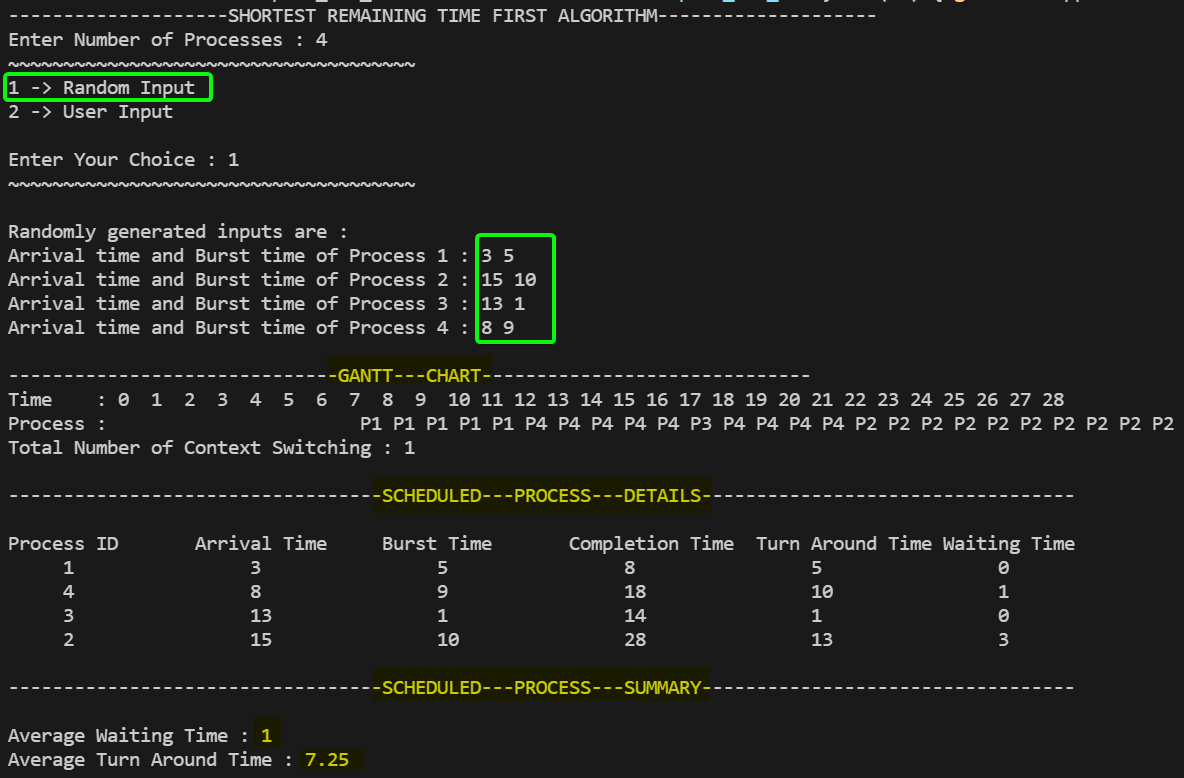
**Output**

|  |  |  |
| --- | --- | --- |
| Process ID | Arrival Time | Burst Time |
| 1 | 3 | 1 |
| 2 | 1 | 4 |
| 3 | 4 | 2 |
| 4 | 0 | 6 |
| 5 | 2 | 3 |

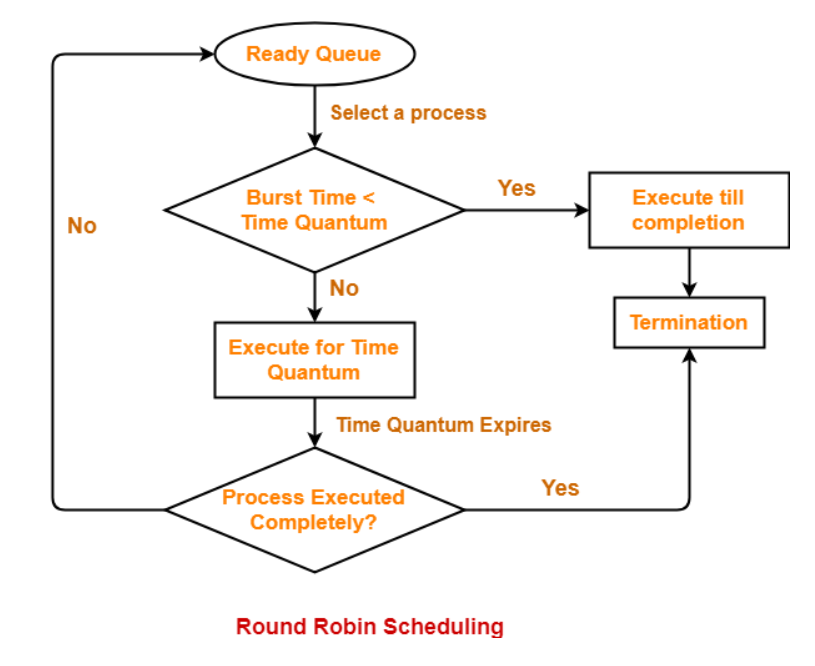




Randomly Generated Input



3. Round Robin Scheduling.



* 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
* The disadvantage of it is **more overhead** of Context Switching.

**Code**

*// Round Robin Algorithm - BHAGYA RANA [U19CS012]*

*#include* <bits/stdc++.h>

using namespace std;

*// "Process" Class*

class Process

{

public:

*// Process ID*

    int id;

*// Arrival Time: Time at which the process arrives in the ready queue.*

    int arrivalTime;

*// Burst Time: Time required by a process for CPU execution.*

    int burstTime;

*// Completion Time: Time at which process completes its execution.*

    int completionTime;

*// Turn Around Time: Time Difference between completion time and arrival time.*

    int turnaroundTime;

*// Waiting Time(W.T): Time Difference between turn around time and burst time.*

    int waitingTime;

    int remainingTime;

*// Constructor for Intializing*

    Process(int id, int arrivalTime, int burstTime)

    {

*this*->id = id;

*this*->arrivalTime = arrivalTime;

*this*->burstTime = burstTime;

*this*->remainingTime = burstTime;

    }

};

*// Compare F(x) for Sorting p*

bool compare(const Process &p1, const Process &p2);

*// F(x) for Round Robin Scheduling*

void roundRobin(vector<Process> &p, int interval);

*// F(x) to Generate Random Input*

vector<Process> randomInputGenerator(int n);

*// F(x) to Print the Output in Well Structured Format*

void printOutput(vector<Process> &p);

*// F(x) to Call the Algorithm*

void solve(vector<Process> &p, int interval);

int main()

{

    cout << "--------------------ROUND ROBIN ALGORITHM--------------------\n";

    int n;

    cout << "Enter Number of Processes : ";

    cin >> n;

    int interval;

    cout << "Enter Time Quantum for Round Robin : ";

    cin >> interval;

    char choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

    cout << "1 -> Random Input\n2 -> User Input\n\n";

    cout << "Enter Your Choice : ";

    cin >> choice;

    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n";

    vector<Process> p;

*if* (choice == '1')

    {

        p = randomInputGenerator(n);

        cout << "Randomly generated inputs are : " << endl;

*for* (int i = 0; i < n; i++)

        {

            cout << "Arrival time and Burst time of Process " << p[i].id << " : " << p[i].arrivalTime << " " << p[i].burstTime << endl;

        }

    }

*else* *if* (choice == '2')

    {

*for* (int i = 0; i < n; i++)

        {

            int arriveTime, burstTime;

            cout << "Enter Arrival time and Burst time of Process " << i + 1 << " : ";

            cin >> arriveTime >> burstTime;

            p.push\_back(Process(i + 1, arriveTime, burstTime));

        }

    }

*else*

    {

        cout << "Incorrect Input Entered\n";

*return* 0;

    }

    solve(p, interval);

*return* 0;

}

*// Customized function for sorting*

bool compare(const Process &p1, const Process &p2)

{

*return* p1.arrivalTime < p2.arrivalTime;

}

*// F(x) for Round Robin Scheduling*

void roundRobin(vector<Process> &p, int interval)

{

*// Count Context Switch*

    int contextSwitch = 0;

    int currTime = 0;

    int prev = -1;

*// Queue to maintain processes.*

    queue<int> q;

    string time = "";

    string processFlow = "";

*// While all processes are not done.*

*while* (true)

    {

        bool flag = true;

*for* (int i = 0; i < p.size(); i++)

        {

*if* (p[i].remainingTime != 0)

            {

                flag = false;

*break*;

            }

        }

*// If all processes are covered.*

*if* (flag)

*break*;

*// If queue is empty, it means that there are no process currently in waiting.*

*// Therefore, jumping to the next process by incrementing currTime.*

*if* (q.empty())

        {

            int prevTime = currTime;

*for* (int i = 0; i < p.size(); i++)

            {

*if* (p[i].remainingTime != 0)

                {

                    currTime = p[i].arrivalTime;

                    q.push(i);

                    int j = i + 1;

*while* (j < p.size() and p[j].arrivalTime == currTime and p[j].remainingTime > 0)

                    {

                        q.push(j);

                        j++;

                    }

*break*;

                }

            }

*for* (int k = prevTime; k < currTime; k++)

                processFlow += " ";

        }

*// Popping the process from queue.*

        int current = q.front();

        q.pop();

*// Incrementing context switch if the previous process is not completed*

*// and also it's not the same as current process*

*if* (prev != -1 and prev != current)

            contextSwitch++;

*// If remaining time is greater than interval then we will minus interval time*

*// and also add processes to queue which arrived in that interval.*

*if* (p[current].remainingTime > interval)

        {

            p[current].remainingTime -= interval;

*for* (int j = current + 1; j < p.size(); j++)

            {

*if* (p[j].arrivalTime > currTime and p[j].arrivalTime <= currTime + interval)

                    q.push(j);

            }

            q.push(current);

            currTime += interval;

            prev = current;

*for* (int k = 0; k < interval; k++)

                processFlow += "P" + to\_string(p[current].id) + " ";

        }

*// If remaining time is lesser than interval then we will make remaining time ZERO*

*// and also add processes to queue which arrived in that time while the process was running.*

*else*

        {

*for* (int j = current + 1; j < p.size(); j++)

            {

*if* (p[j].arrivalTime > currTime and p[j].arrivalTime <= currTime + p[current].remainingTime)

                    q.push(j);

            }

*for* (int k = 0; k < p[current].remainingTime; k++)

                processFlow += "P" + to\_string(p[current].id) + " ";

            currTime += p[current].remainingTime;

*// Updating records for a process which is completed.*

            p[current].remainingTime = 0;

            p[current].completionTime = currTime;

            p[current].turnaroundTime = currTime - p[current].arrivalTime;

            p[current].waitingTime = currTime - p[current].burstTime - p[current].arrivalTime;

            prev = -1;

        }

    }

*for* (int k = 0; k < currTime; k++)

    {

        time += (to\_string(k) + " ");

*if* (k < 10)

            time += " ";

    }

    cout << "\n------------------------------GANTT---CHART------------------------------\n\n";

    cout << "Time : " + time << endl;

    cout << "Process : " + processFlow << endl;

    cout << "Total Number of Context Switching : " << contextSwitch << endl;

}

*// Function to generate random input*

vector<Process> randomInputGenerator(int n)

{

    unsigned seed = chrono::system\_clock::now().time\_since\_epoch().count();

    default\_random\_engine generator(seed);

    uniform\_int\_distribution<int> d1(0, ((10 \* n) + 1) / 2);

    uniform\_int\_distribution<int> d2(1, 10);

    vector<Process> p;

*for* (int i = 0; i < n; i++)

    {

        p.push\_back(Process(i + 1, d1(generator), d2(generator)));

        d1.reset();

    }

*return* p;

}

*// Referrence : https://stackoverflow.com/questions/14765155/how-can-i-easily-format-my-data-table-in-c*

template <typename T>

void printElement(T t)

{

    cout << left << setw(17) << setfill(' ') << t;

}

*// Utility function to print Output*

void printOutput(vector<Process> &p)

{

    int TotalWaiting = 0, TotalTurnAround = 0;

    cout << "\n----------------------------------SCHEDULED---PROCESS---DETAILS----------------------------------\n\n";

    printElement("Process ID");

    printElement("Arrival Time");

    printElement("Burst Time");

    printElement("Completion Time");

    printElement("Turn Around Time");

    printElement("Waiting Time");

    cout << endl;

*for* (int i = 0; i < p.size(); i++)

    {

        cout << "     ";

        printElement(p[i].id);

        printElement(p[i].arrivalTime);

        printElement(p[i].burstTime);

        printElement(p[i].completionTime);

        printElement(p[i].turnaroundTime);

        printElement(p[i].waitingTime);

        cout << endl;

        TotalWaiting += p[i].waitingTime;

        TotalTurnAround += p[i].turnaroundTime;

    }

    cout << "\n----------------------------------SCHEDULED---PROCESS---SUMMARY----------------------------------\n\n";

    cout << "Average Waiting Time : " << (double)TotalWaiting / (double)p.size() << endl;

    cout << "Average Turn Around Time : " << (double)TotalTurnAround / (double)p.size() << endl;

}

*// Function which will call other needed functions.*

void solve(vector<Process> &p, int interval)

{

    sort(p.begin(), p.end(), compare);

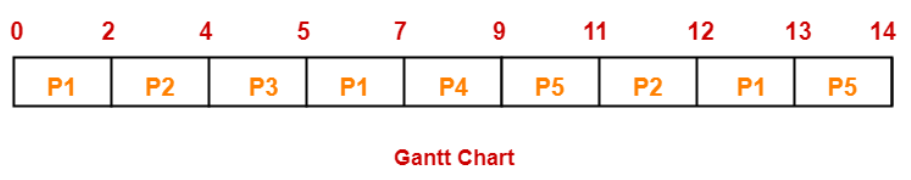
    roundRobin(p, interval);

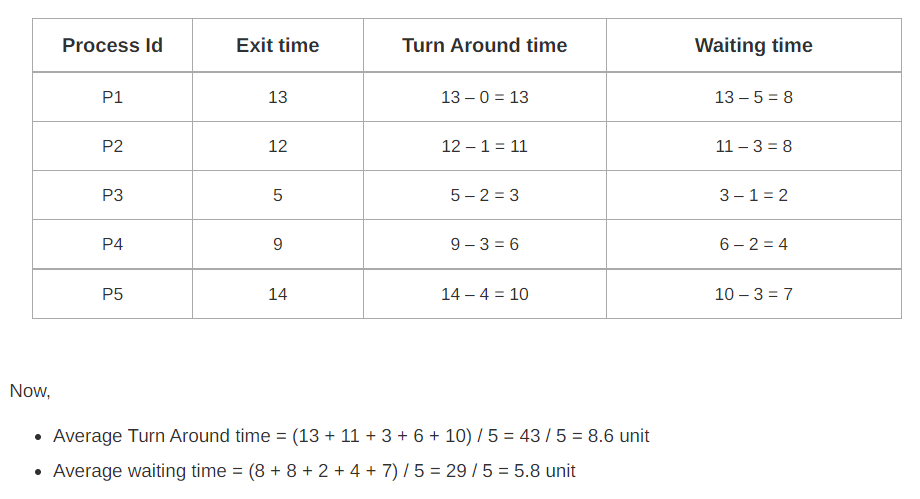
    printOutput(p);

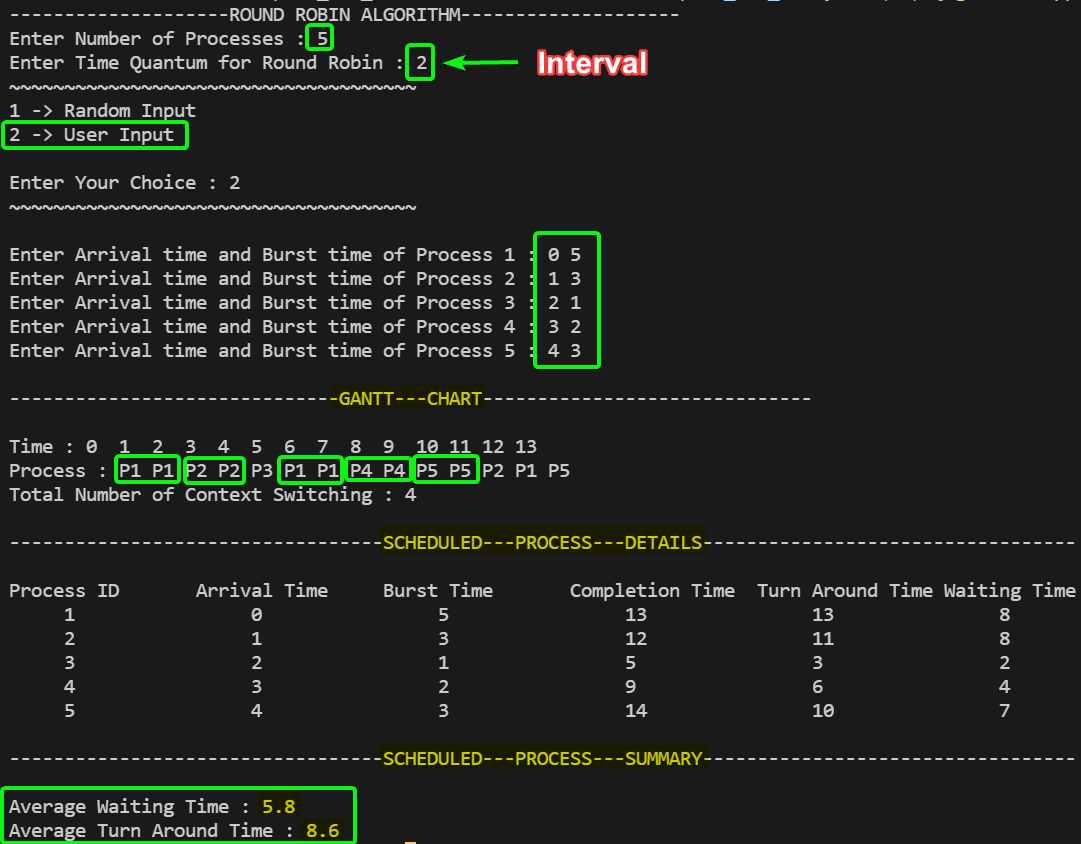
}

**Output**

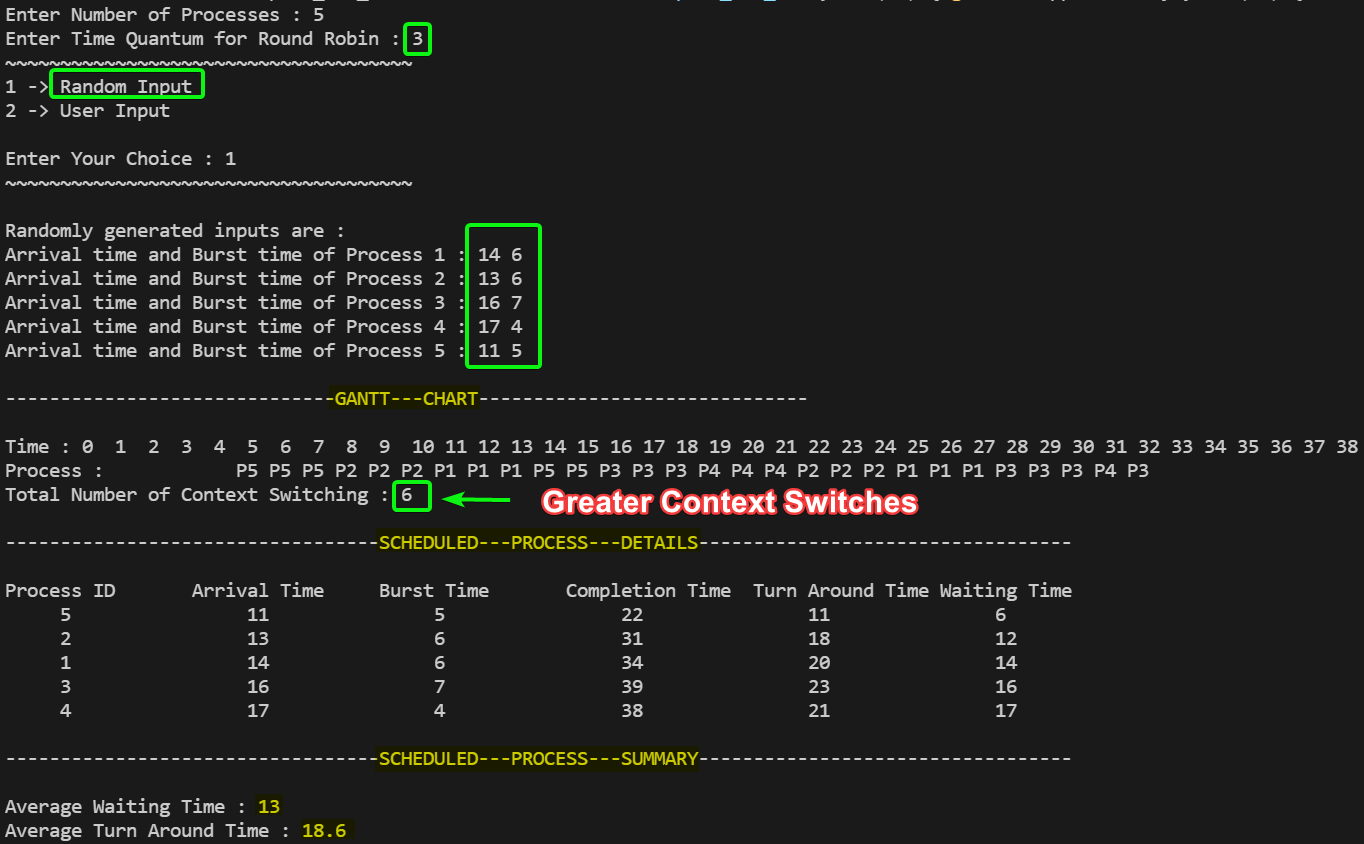
|  |  |  |
| --- | --- | --- |
| Process ID | Arrival Time | Burst Time |
| 1 | 0 | 5 |
| 2 | 1 | 3 |
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 4 | 3 |







Randomly Generated Input



Thus, we have Successfully Understood and **Implemented**

* Shortest Job First (SJF) [Non Preemptive]
* Shortest Remaining Time First (SRTF) [Preemptive SJF]
* Round Robin Scheduling

SUBMITTED BY:

**U19CS012**

BHAGYA VINOD RANA