Project report

TOPIC: CPU Scheduling

Sumbmitted by:

Rohit kumar

4th year, EE

21115121

Table of contents:

1 . Introduction:

In modern computing systems, the efficiency and responsiveness of an operating system largely depend on its ability to manage and schedule processes effectively. CPU scheduling is a crucial component of process management, ensuring that multiple processes can share the CPU's time in an optimal manner. The choice of scheduling algorithm can significantly impact system performance, influencing key metrics such as waiting time, turnaround time, CPU utilization, and throughput.

Several CPU scheduling algorithms have been developed to address various system requirements and workloads. Each algorithm employs a unique strategy for selecting which process to execute next. These strategies range from simple first-come, first-served approaches to more complex priority-based schemes. Understanding the strengths and weaknesses of each algorithm is essential for selecting the most suitable one for a given set of processes and system conditions.

2. Objective:

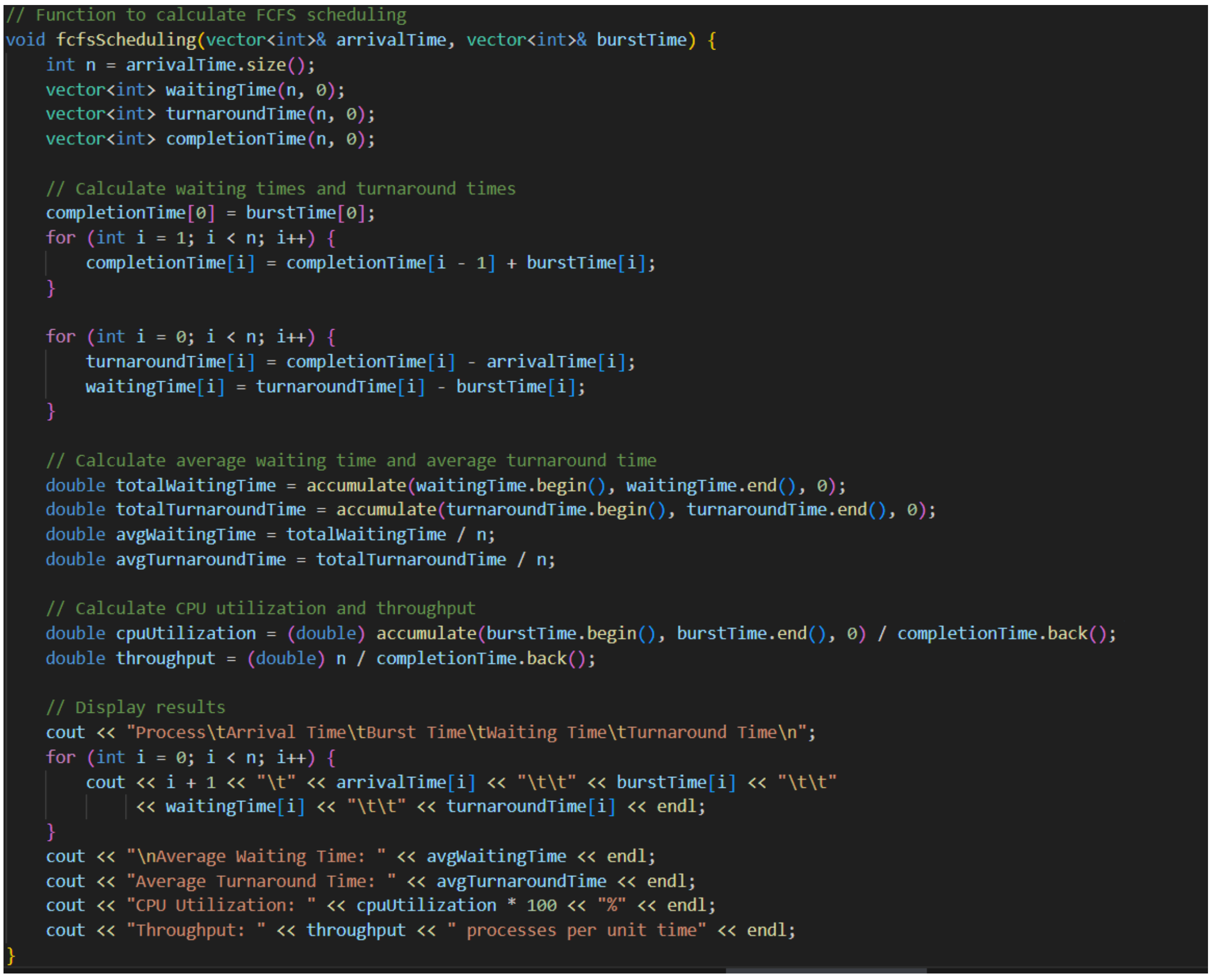
The objective of this project is to implement and compare four widely used CPU scheduling algorithms in C++: First-come-first-served (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), and Round Robin (RR). By providing a set of processes with specific arrival times and burst times, the project aims to determine which algorithm performs best under the given conditions. The comparison is based on several critical performance metrics: average waiting time, average turnaround time, CPU utilization, and throughput.

The implementation involves the development of a comprehensive program that simulates the execution of these scheduling algorithms. The program, upon receiving the input of arrival times and burst times for a set of processes, calculates and compares the performance metrics for each algorithm. The ultimate goal is to identify the algorithm that offers the most optimal performance for the given set of processes, thereby providing insights into which scheduling method is most efficient in various scenarios. This structured approach ensures a systematic and thorough comparison of the algorithms.

3. Scheduling algorithms implemented:

3.1 First -come, first-served basis(FEFS) :

First-Come, First-Served (FCFS) is a straightforward CPU scheduling algorithm where the first process to request the CPU is the first to receive it. This non-preemptive method follows a FIFO (First-In-First-Out) approach, treating processes like a queue at a service counter. It is simple and fair, ensuring that processes are handled in the order they arrive. Despite its limitations, FCFS is easy to implement and ensures no process starvation, making it suitable for batch systems where simplicity and fairness are prioritized over fast response times. Here is the simple C++ code for its implementation:



```cpp
// Function to calculate FCFS scheduling
void fcfsScheduling(vector<int>& arrivalTime, vector<int>& burstTime) {
    int n = arrivalTime.size();
    vector<int> waitingTime(n, 0);
    vector<int> turnaroundTime(n, 0);
    vector<int> completionTime(n, 0);

    // Calculate waiting times and turnaround times
    completionTime[0] = burstTime[0];
    for (int i = 1; i < n; i++) {
        completionTime[i] = completionTime[i - 1] + burstTime[i];
    }

    for (int i = 0; i < n; i++) {
        turnaroundTime[i] = completionTime[i] - arrivalTime[i];
        waitingTime[i] = turnaroundTime[i] - burstTime[i];
    }

    // Calculate average waiting time and average turnaround time
    double totalWaitingTime = accumulate(waitingTime.begin(), waitingTime.end(), 0);
    double totalTurnaroundTime = accumulate(turnaroundTime.begin(), turnaroundTime.end(), 0);
    double avgWaitingTime = totalWaitingTime / n;
    double avgTurnaroundTime = totalTurnaroundTime / n;

    // Calculate CPU utilization and throughput
    double cpuUtilization = (double) accumulate(burstTime.begin(), burstTime.end(), 0) / completionTime.back();
    double throughput = (double) n / completionTime.back();

    // Display results
    cout << "Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";
    for (int i = 0; i < n; i++) {
        cout << i + 1 << "\t" << arrivalTime[i] << "\t\t" << burstTime[i] << "\t\t"
             << waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }
    cout << "\nAverage Waiting Time: " << avgWaitingTime << endl;
    cout << "Average Turnaround Time: " << avgTurnaroundTime << endl;
    cout << "CPU Utilization: " << cpuUtilization * 100 << "%" << endl;
    cout << "Throughput: " << throughput << " processes per unit time" << endl;
}
```

3.2 Shortest job first(SJF) algorithm:

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the shortest burst time for execution next. This approach can be either preemptive or non-preemptive. SJF minimizes the average waiting time compared to other algorithms by prioritizing shorter tasks, leading to improved overall system performance. This algorithm is optimal when all process burst times are known in advance. Here below simple c++ function for implementing this algorithm:

```cpp
void sjf_scheduling(vector<int> arrival_times, vector<int> burst_times,
vector<int> process_ids) {
    int n = arrival_times.size();
    vector<int> completion_times(n, 0), waiting_times(n,
0),turnaround_times(n, 0);
   vector<int> response_times(n, 0),start_times(n, -1),remaining_processes(n);
    vector<bool> started(n, false);
    for (int i = 0; i < n; ++i) {
        remaining_processes[i] = i;
    }
    int current_time = 0, completed = 0,total_burst_time = 0;
```

```
     double total_waiting_time = 0,total_turnaround_time =
0,total_response_time = 0;;

    while (completed < n) {
        vector<int> available_processes;
        for (int i : remaining_processes) {
            if (arrival_times[i] <= current_time && !started[i]) {
                available_processes.push_back(i);
            }
        }

        if (available_processes.empty()) {
            current_time++;
            continue;
        }

        auto shortest_job = min_element(available_processes.begin(),
available_processes.end(), [&burst_times](int a, int b) {
            return burst_times[a] < burst_times[b];
        });

        int job_index = *shortest_job;
        start_times[job_index] = current_time;
        completion_times[job_index] = current_time + burst_times[job_index];
        waiting_times[job_index] = start_times[job_index] -
arrival_times[job_index];
        turnaround_times[job_index] = completion_times[job_index] -
arrival_times[job_index];
        response_times[job_index] = start_times[job_index] -
arrival_times[job_index];
        started[job_index] = true;

        total_waiting_time += waiting_times[job_index];
        total_turnaround_time += turnaround_times[job_index];
        total_response_time += response_times[job_index];
        total_burst_time += burst_times[job_index];

        current_time = completion_times[job_index];
        completed++;
    }

    double avg_waiting_time = total_waiting_time / n;
    double avg_turnaround_time = total_turnaround_time / n;
    double avg_response_time = total_response_time / n;
    double throughput = static_cast<double>(n) / current_time;
    double cpu_utilization = (total_burst_time /
static_cast<double>(current_time)) * 100;
```

```
    cout << "Process ID | Arrival Time | Burst Time | Completion Time | 
Waiting Time | Turnaround Time | Response Time" << endl;
    for (int i = 0; i < n; ++i) {
        cout << setw(10) << process_ids[i] << " | "
             << setw(12) << arrival_times[i] << " | "
             << setw(10) << burst_times[i] << " | "
             << setw(15) << completion_times[i] << " | "
             << setw(12) << waiting_times[i] << " | "
             << setw(15) << turnaround_times[i] << " | "
             << setw(13) << response_times[i] << endl;
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Waiting Time: " << avg_waiting_time << endl;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
    cout << "Average Response Time: " << avg_response_time << endl;
    cout << "Throughput: " << throughput << " processes/unit time" << endl;
    cout << "CPU Utilization: " << cpu_utilization << "%" << endl;
}
```

3.3 <u>Shortest remaing time first(SRTF) algorithm: </u>Shortest Remaining Time First (SRTF) is a preemptive CPU scheduling algorithm that selects the process with the shortest remaining burst time for execution. Whenever a new process arrives with a shorter burst time than the currently running process, the CPU switches to the new process. This approach aims to minimize the average waiting time and improve system responsiveness by prioritizing shorter tasks. SRTF is most effective when precise knowledge of process burst times is available.Here is a c++ code for its implementation:

```
void srtfScheduling(const vector<int>& arrivalTime, const vector<int>& 
burstTime) {
    int n = arrivalTime.size();
    vector<int> remainingTime = burstTime;
    vector<int> waitingTime(n, 0);
    vector<int> turnaroundTime(n, 0);
    vector<int> completionTime(n, 0);

    int currentTime = 0;
    int completed = 0;
    int shortest = 0;
    int minRemainingTime = numeric_limits<int>::max();
    bool foundProcess = false;

    while (completed != n) {
        // Find the process with the smallest remaining time at the current 
time
```

```
        for (int i = 0; i < n; i++) {
            if (arrivalTime[i] <= currentTime && remainingTime[i] > 0 &&
remainingTime[i] < minRemainingTime) {
                minRemainingTime = remainingTime[i];
                shortest = i;
                foundProcess = true;
            }
        }

        if (!foundProcess) {
            currentTime++;
            continue;
        }

        // Decrement the remaining time of the selected process
        remainingTime[shortest]--;
        minRemainingTime = remainingTime[shortest];

        if (minRemainingTime == 0) {
            minRemainingTime = numeric_limits<int>::max();
        }

        // If a process gets completely executed
        if (remainingTime[shortest] == 0) {
            completed++;
            foundProcess = false;
            int finishTime = currentTime + 1;
            completionTime[shortest] = finishTime;
            turnaroundTime[shortest] = finishTime - arrivalTime[shortest];
            waitingTime[shortest] = turnaroundTime[shortest] -
burstTime[shortest];
        }

        currentTime++;
    }
```

3.4 Round-Robbin (RR) algorithm:

Round Robin (RR) is a non-preemptive CPU scheduling algorithm designed to distribute CPU time fairly among processes. Each process is assigned a fixed time slice or quantum, and processes are executed in a circular order. When a process's time quantum expires, it is placed at the end of the ready queue, and the next process is given the CPU. This approach ensures that all processes receive an equal share of CPU time, reducing the chance of process starvation and improving responsiveness in time-sharing systems. However, the performance of RR heavily depends on the chosen time quantum: too large a quantum can lead to poor response times, while too small a quantum can cause excessive context switching and overhead. Here is a c++ code for its implementation:

```
void roundRobinScheduling(vector<int>& arrivalTime, vector<int>& burstTime,
int timeQuantum) {
    int n = arrivalTime.size();
    vector<int> remainingTime = burstTime;
    vector<int> waitingTime(n, 0);
    vector<int> turnaroundTime(n, 0);
    vector<int> completionTime(n, 0);

    queue<int> processQueue;
    int currentTime = 0;
    int completed = 0;

    // Enqueue processes as they arrive
    int index = 0;
    while (completed != n) {
        while (index < n && arrivalTime[index] <= currentTime) {
            processQueue.push(index);
            index++;
        }

        if (processQueue.empty()) {
            currentTime++;
            continue;
        }

        int currentProcess = processQueue.front();
        processQueue.pop();

        int timeToRun = min(timeQuantum, remainingTime[currentProcess]);
        currentTime += timeToRun;
        remainingTime[currentProcess] -= timeToRun;

        while (index < n && arrivalTime[index] <= currentTime) {
            processQueue.push(index);
            index++;
        }

        if (remainingTime[currentProcess] > 0) {
            processQueue.push(currentProcess);
        } else {
            completed++;
            completionTime[currentProcess] = currentTime;
            turnaroundTime[currentProcess] = completionTime[currentProcess] -
arrivalTime[currentProcess];
            waitingTime[currentProcess] = turnaroundTime[currentProcess] -
burstTime[currentProcess];
        }
    }
```

```cpp
    // Calculate CPU utilization and throughput
    int totalBurstTime = accumulate(burstTime.begin(), burstTime.end(), 0);
    double cpuUtilization = (double) totalBurstTime / currentTime;
    double throughput = (double) n / currentTime;

    // Calculate average waiting time and average turnaround time
    double totalWaitingTime = accumulate(waitingTime.begin(),
waitingTime.end(), 0);
    double totalTurnaroundTime = accumulate(turnaroundTime.begin(),
turnaroundTime.end(), 0);
    double avgWaitingTime = totalWaitingTime / n;
    double avgTurnaroundTime = totalTurnaroundTime / n;

    // Display results
    cout << "Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n";
    for (int i = 0; i < n; i++) {
        cout << i + 1 << "\t\t" << arrivalTime[i] << "\t\t" << burstTime[i] <<
"\t\t"
             << waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }
    cout << "\nAverage Waiting Time: " << avgWaitingTime << endl;
    cout << "Average Turnaround Time: " << avgTurnaroundTime << endl;
    cout << "CPU Utilization: " << cpuUtilization * 100 << "%" << endl;
    cout << "Throughput: " << throughput << " processes per unit time" <<
endl;
}
```

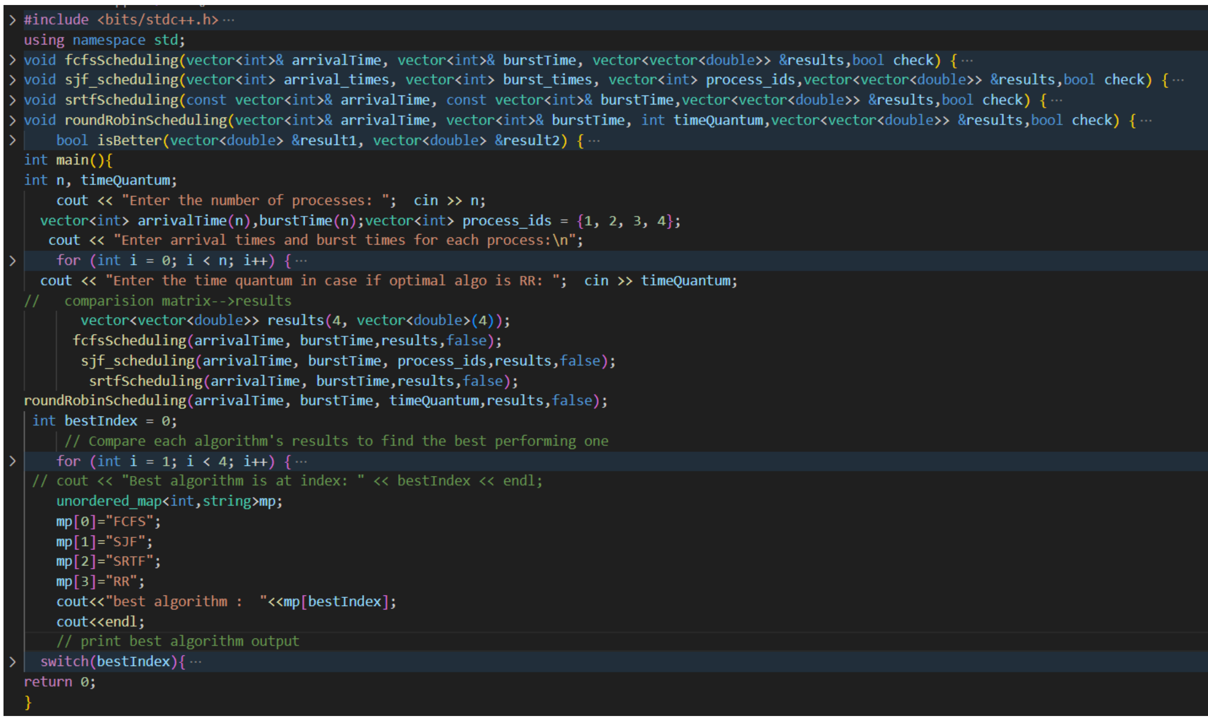
4. Implementation details:

4.1 Tools Used

- **Programming Language**: The project is implemented using C++, a powerful language that provides fine-grained control over system resources and performance.
- **Integrated Development Environment (IDE)**: The development and debugging were carried out using Visual Studio and Code.

4.2 Main Program(thought process) :

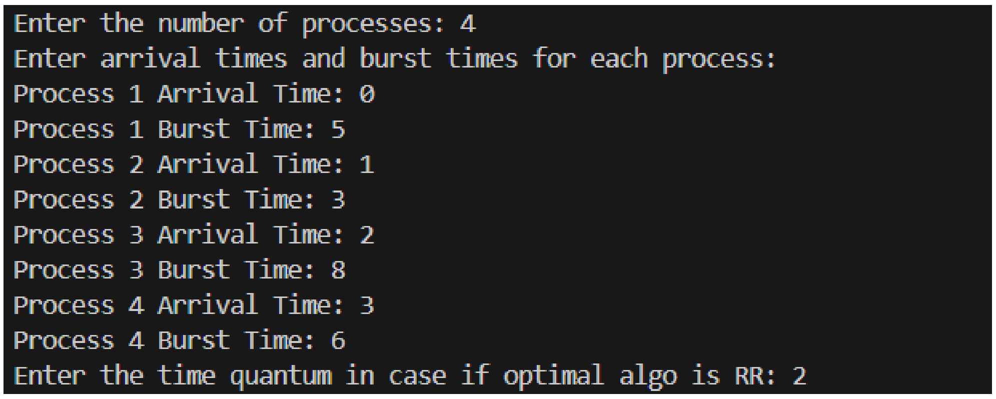
My program implements and evaluates four fundamental CPU scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), and Round Robin (RR). Each algorithm is designed to manage processes efficiently by prioritizing tasks based on arrival times and burst durations. The project begins by collecting user input, including process details and scheduling parameters. The implementation includes dedicated functions for each algorithm, computing key metrics such as average waiting time, average turnaround time, CPU utilization, and throughput. These metrics serve as benchmarks for comparing algorithm performance. The `isBetter` function facilitates this comparison by prioritizing algorithms based on metrics hierarchy: waiting time, turnaround time, CPU utilization, and throughput. Results are compiled into a comparison matrix, allowing for a systematic evaluation of each algorithm's effectiveness under varying workloads. The optimal algorithm is determined based on which achieves the best overall performance across these metrics. Here is the c++ code file of the main program:



```cpp
> #include <bits/stdc++.h> ···
using namespace std;
> void fcfsScheduling(vector<int>& arrivalTime, vector<int>& burstTime, vector<vector<double>> &results,bool check) {···
> void sjf_scheduling(vector<int> arrival_times, vector<int> burst_times, vector<int> process_ids,vector<vector<double>> &results,bool check) {···
> void srtfScheduling(const vector<int>& arrivalTime, const vector<int>& burstTime,vector<vector<double>> &results,bool check) {···
> void roundRobinScheduling(vector<int>& arrivalTime, vector<int>& burstTime, int timeQuantum,vector<vector<double>> &results,bool check) {···
>     bool isBetter(vector<double> &result1, vector<double> &result2) {···
int main(){
int n, timeQuantum;
    cout << "Enter the number of processes: ";  cin >> n;
  vector<int> arrivalTime(n),burstTime(n);vector<int> process_ids = {1, 2, 3, 4};
   cout << "Enter arrival times and burst times for each process:\n";
>   for (int i = 0; i < n; i++) {···
  cout << "Enter the time quantum in case if optimal algo is RR: ";  cin >> timeQuantum;
//   comparision matrix-->results
       vector<vector<double>> results(4, vector<double>(4));
      fcfsScheduling(arrivalTime, burstTime,results,false);
       sjf_scheduling(arrivalTime, burstTime, process_ids,results,false);
        srtfScheduling(arrivalTime, burstTime,results,false);
roundRobinScheduling(arrivalTime, burstTime, timeQuantum,results,false);
 int bestIndex = 0;
     // Compare each algorithm's results to find the best performing one
>    for (int i = 1; i < 4; i++) {···
 // cout << "Best algorithm is at index: " << bestIndex << endl;
    unordered_map<int,string>mp;
    mp[0]="FCFS";
    mp[1]="SJF";
    mp[2]="SRTF";
    mp[3]="RR";
    cout<<"best algorithm :  "<<mp[bestIndex];
    cout<<endl;
    // print best algorithm output
>  switch(bestIndex){···
return 0;
}
```

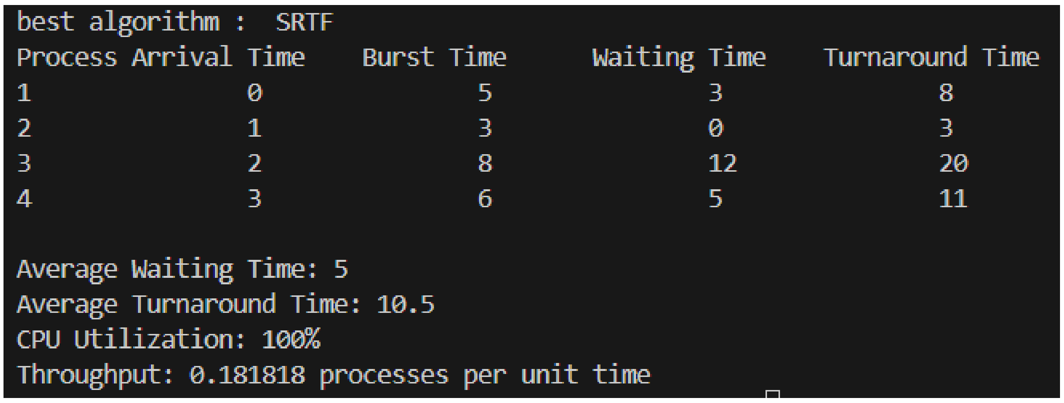
4.3 main program workflow

1. **Input Handling**:
   - The program prompts the user to input the number of processes.
   - For each process, it gathers the arrival time and burst time, storing them in vectors.



```
Enter the number of processes: 4
Enter arrival times and burst times for each process:
Process 1 Arrival Time: 0
Process 1 Burst Time: 5
Process 2 Arrival Time: 1
Process 2 Burst Time: 3
Process 3 Arrival Time: 2
Process 3 Burst Time: 8
Process 4 Arrival Time: 3
Process 4 Burst Time: 6
Enter the time quantum in case if optimal algo is RR: 2
```

2. **Executing Scheduling Algorithms**:
   - The program calls each scheduling algorithm function (FCFS, SJF, SRTF, RR).
   - Each function processes the list of processes, calculates performance metrics, and stores these metrics in a results matrix.
3. **Creating Comparison Matrix**:
   - The results from each algorithm (average waiting time, average turnaround time, CPU utilization, and throughput) are stored in a comparison matrix(results matrix).
4. **Determining Optimal Algorithm**:
   - The comparison matrix is passed to the isBetter function.
   - The isBetter function compares the metrics and determines which algorithm is the most optimal based on predefined criteria.
5. **Outputting Final Metrics**:
   - The program outputs the metrics of the optimal algorithm, displaying the average waiting time, average turnaround time, CPU utilization, and throughput for the best-performing algorithm.



```
best algorithm :  SRTF
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1           0               5               3               8
2           1               3               0               3
3           2               8               12              20
4           3               6               5               11

Average Waiting Time: 5
Average Turnaround Time: 10.5
CPU Utilization: 100%
Throughput: 0.181818 processes per unit time
```

5. Comparision matrix matrices for comparison:

5.1. **Average Waiting Time**:

- The average amount of time a process spends waiting in the ready queue before getting CPU execution.
- Sum of waiting times of all processes divided by the number of processes.

- Lower waiting times indicate better performance as processes spend less time waiting and more time executing.

5.2. **Average Turnaround Time**:

- The average time taken for a process to complete execution, i.e., the time from arrival to completion.
- Sum of turnaround times of all processes divided by the number of processes.
- Lower turnaround times signify faster completion of processes and better overall efficiency.

5.3. **CPU Utilization**:

- The percentage of time the CPU is executing processes rather than idling.
- Total CPU time spent on executing processes divided by the total time.
- Higher CPU utilization indicates efficient use of CPU resources and maximization of processing capacity.

5.4. **Throughput**:

- The number of processes completed per unit of time.
- Total number of processes divided by the total time taken to execute them.
- Higher throughput indicates how many processes are completed within a given time frame, reflecting system efficiency.

6. Comparison of different algorithm to find optimal one:

In my study, i aim to identify the optimal scheduling algorithm based on several key performance metrics. To facilitate this comparison, i have implemented a function, `isBetter`, which compares two algorithms and determines which one performs better. The metrics considered for this comparison are average waiting time, average turnaround time, and CPU utilization. This function is a valuable tool in my analysis, enabling clear and consistent comparisons across different scheduling strategies.The isbetter function is defined as follows:

```
bool isBetter(vector<double> &result1, vector<double> &result2) {
  // Compare based on average waiting time
  if (result1[0] < result2[0]) {
      return true;
  } else if (result1[0] > result2[0]) {
      return false;
  }

  // If waiting times are equal, compare based on average turnaround time
  if (result1[1] < result2[1]) {
      return true;
  } else if (result1[1] > result2[1]) {
        return false;
    }

    // If turnaround times are equal, compare based on CPU utilization
    if (result1[3] > result2[3]) {
        return true;
    } else if (result1[3] < result2[3]) {
        return false;
    }

    // If all metrics are equal, return false (they are equivalent in
performance)
    return false;
}
```

7. Summary:

This project implements and evaluates four CPU scheduling algorithms: FCFS, SJF, SRTF, and RR. It calculates key performance metrics, including average waiting time, average turnaround time, CPU utilization, and throughput for each algorithm. By comparing these metrics, the project identifies the optimal algorithm for a given set of processes. The best-performing algorithm is determined based on minimizing waiting and turnaround times while maximizing CPU utilization and throughput. This approach provides insights into the most efficient scheduling strategy, enhancing system performance and responsiveness.