CPU Scheduling Algorithms

This repository contains implementations of various CPU scheduling algorithms. Each algorithm has been implemented in C++ and the performance metrics such as average waiting time and average turn-around time are calculated and displayed. This document explains each scheduling algorithm in detail and outlines how the predicted algorithm is determined based on the input processes.

Table of Contents

Basic Terminologies

- **Arrival Time**: The time at which the process arrives in the ready queue.
- **Completion Time**: The time at which the process completes its execution.
- **Burst Time**: The time required by a process for CPU execution.
- **Turn Around Time (TAT)**: The time difference between the completion time and the arrival time.
    - **Formula**: `Turn Around Time = Completion Time - Arrival Time`
- **Waiting Time (WT)**: The time difference between the turn-around time and the burst time.
    - **Formula**: `Waiting Time = Turn Around Time - Burst Time`

Algorithms Implemented

The following CPU scheduling algorithms have been implemented:

1. First-Come, First-Served (FCFS)
2. Shortest Job First (SJF)
3. Longest Job First (LJF)
4. Shortest Remaining Time First (SRTF)
5. Round Robin (RR)

Details of Each Algorithm

First-Come, First-Served (FCFS)

FCFS is considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics of FCFS:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS:

- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

- FCFS suffers from Convoy effect.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

Explanation of Code:

Here, the code is implemented using a vector containing all the values defining a process and we sort the vector in increasing order of arrival times. Then, each and every processes comes one after the another and gets executed.

```
vector<vector<float>> FCFS(vector<pair<int, pair<float, float>>> processes, ofstream& outputFile) {
   vector<pair<int, pair<float, float>>> original(processes.begin(), processes.end());
   sort(processes.begin(), processes.end(), IncArrTime);
   vector<vector<float>> run;
   float RunTime = 0;

   for (auto& p : processes) {
       float pID = p.first;
       float arrTime = p.second.first;
       float burstTime = p.second.second;

       if (RunTime < arrTime) {
           RunTime = arrTime;
       }

       float waitTime = max(0.0f, RunTime - arrTime);
       float TATime = RunTime + burstTime;

       run.push_back({pID, RunTime, original[pID - 1].second.second, original[pID - 1].second.first, TATime});
       RunTime = TATime;
   }

   outputFile << "FCFS :\n";
   GanttChart(run, outputFile);
   return run;
}
```

Shortest Job First (SJF)

Shortest Job First (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive and significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Characteristics of SJF:

- Shortest Job First has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages of Shortest Job First:

- As SJF reduces the average waiting time, it is better than the First Come First Serve scheduling algorithm.
- SJF is generally used for long-term scheduling.

Disadvantages of SJF:

- One of the demerits SJF has is starvation.
- Many times it becomes complicated to predict the length of the upcoming CPU request.

Explanation of Code:

Here, a multiset is being used so that the sorting of processes, according to their burst times, comes handy and since this algorithm is non-preemptive, we are checking for new values at the completion of the process only and not in between.

```
vector< vector<float>> SJF(vector< pair<int, pair<float, float>>> processes, ofstream& outputFile){
    vector< pair<int, pair<float, float>>> original(processes.begin(), processes.end());
    sort(processes.begin(), processes.end(), IncArrTime);

    multiset <pair<float, pair<float, int>>> ReadyQueue;
    vector< vector<float>> run;

    int idx = 0;
    float RunTime = 0;
    int totalProcesses = processes.size();

    while(idx < totalProcesses || !ReadyQueue.empty()){

        while(idx < totalProcesses && processes[idx].second.first <= RunTime){
            ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if(ReadyQueue.empty()){
            RunTime = processes[idx].second.first;
            continue;
        }

        pair<float, pair<float, int>> RunningProcess = *ReadyQueue.begin();
        ReadyQueue.erase(ReadyQueue.begin());

        float pID = RunningProcess.second.second;
        float arrTime = RunningProcess.second.first;
        float burstTime = RunningProcess.first;

        float waitTime = max(0.0f, RunTime - arrTime);
        float TATime = RunTime + burstTime;

        run.push_back({pID, RunTime, original[pID-1].second.second, original[pID-1].second.first, TATime});
        RunTime += burstTime;
    }
    outputFile << "SJF :\n";
    GanttChart(run, outputFile);
    return run;
}
```

Longest Job First (LJF)

Longest Job First (LJF) scheduling process is the opposite of Shortest Job First (SJF). This algorithm prioritizes the process with the largest burst time, meaning the longest job is processed first. LJF is non-preemptive in nature.

Characteristics of LJF:

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.
- If two processes have the same burst time, then the tie is broken using FCFS (First Come First Serve) i.e., the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of LJF:

- No other task can be scheduled until the longest job or process executes completely.
- All the jobs or processes finish at approximately the same time.

Disadvantages of LJF:

- Generally, the LJF algorithm gives a very high average waiting time and average turnaround time for a given set of processes.
- This may lead to the convoy effect, where short processes wait for a long process to finish.

Explanation of Code:

This implementation is also very similar to that of SJF. The only difference is that, a `struct` is introduced to deal with the decreasing order of burst times.

```
struct DecSet {
    bool operator()(const pair<float, pair<float, int>>& a, const pair<float, pair<float, int>>& b) const {
        return a.first > b.first;
    }
};

vector< vector<float>> LJF(vector< pair<int, pair<float, float>>> processes, ofstream& outputFile) {
    vector< pair<int, pair<float, float>>> original(processes.begin(), processes.end());
    sort(processes.begin(), processes.end(), IncArrTime);

    multiset< pair<float, pair<float, int>>, DecSet> ReadyQueue;
    vector< vector<float>> run;

    int idx = 0;
    float RunTime = 0;
    int totalProcesses = processes.size();

    while (idx < totalProcesses || !ReadyQueue.empty()) {
        while (idx < totalProcesses && processes[idx].second.first <= RunTime) {
            ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if (ReadyQueue.empty()) {
            if (idx < totalProcesses) {
                RunTime = processes[idx].second.first;
            }
            continue;
        }

        auto RunningProcess = *ReadyQueue.begin();
        ReadyQueue.erase(ReadyQueue.begin());

        float pID = RunningProcess.second.second;
        float arrTime = RunningProcess.second.first;
        float burstTime = RunningProcess.first;

        float waitTime = max(0.0f, RunTime - arrTime);
        float TATime = RunTime + burstTime;

        run.push_back({pID, RunTime, original[pID - 1].second.second, original[pID - 1].second.first, TATime});
        RunTime += burstTime;
    }

    outputFile << "LJF :\n";
    GanttChart(run, outputFile);
    return run;
}
```

Shortest Remaining Time First (SRTF)

Shortest Remaining Time First (SRTF) is the preemptive version of Shortest Job First (SJF), where the processor is allocated to the job closest to completion. In SRTF, the process with the smallest amount of time remaining until completion is selected to execute.

Characteristics of Shortest Remaining Time First:

- SRTF algorithm speeds up the processing of jobs compared to SJF, assuming overhead charges are not considered.
- Context switches occur more frequently in SRTF than in SJF, consuming CPU time, which can diminish its advantage of fast processing.

Advantages of SRTF:

- Short processes are handled very quickly in SRTF.
- The system requires minimal overhead since decisions are made only when a process completes or a new process arrives.

Disadvantages of SRTF:

- Similar to SJF, SRTF has the potential for process starvation.

- Long processes may be delayed indefinitely if short processes continually arrive.

Explanation of Code:

As it is already stated that this algorithm is just preemptive version of SJF. So, here at the end of every `second`, the code checks for new processes (if any) and updates the multiset in order to make the task with least burst time available to CPU.

```
vector< vector<float>> SRTF(vector< pair<int, pair<float, float>>> processes, ofstream& outputFile){
    vector< pair<int, pair<float, float>>> original(processes.begin(), processes.end());
    sort(processes.begin(), processes.end(), IncArrTime);

    multiset < pair<float, pair<float, int>>> ReadyQueue;
    vector< vector<float>> run, exec;

    int idx = 0, n = processes.size();
    float RunTime = 0;

    ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
    RunTime = max(RunTime, processes[idx].second.first);
    idx++;

    while(!ReadyQueue.empty() || idx < n){
        while(idx < n && processes[idx].second.first <= RunTime){
            ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if (ReadyQueue.empty()) {
            RunTime = processes[idx].second.first;
            ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        pair<float, pair<float, int>> RunningProcess = *ReadyQueue.begin();
        ReadyQueue.erase(ReadyQueue.begin());

        float pID = RunningProcess.second.second;
        float arrTime = RunningProcess.second.first;
        float burstTime = RunningProcess.first;
        float remTime = burstTime - 1;

        float waitTime = max(0.0f, RunTime - arrTime);
        float TATime = RunTime + 1;

        exec.push_back({pID, RunTime, 1.0f, waitTime, RunTime+1});

        RunTime++;

        while(idx < n && processes[idx].second.first <= RunTime){
            ReadyQueue.insert({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if(remTime == 0){
            run.push_back({pID, arrTime, original[pID-1].second.second, original[pID-1].second.first, TATime});
        } else {
            ReadyQueue.insert({remTime, {arrTime, pID}});
        }
    }
    outputFile << "SRTF :\n";
    GanttChart(exec, outputFile);
    return run;
}
```

Round Robin (RR)

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of the First Come First Serve CPU Scheduling algorithm and generally focuses on time-sharing techniques.

Characteristics of Round Robin:

- It's simple, easy to use, and starvation-free as all processes get balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as processes are given CPU time for a very limited duration.

Advantages of Round Robin:

- Round Robin is fair as every process gets an equal share of CPU time.
- Newly created processes are added to the end of the ready queue.

Explanation of Code:

The code of round robin algorithm has a quantum value of `2s` and it checks for new processes coming before or at the time of preemption (or complemetion), whichever is applicable, then moves the current process `(if burst time remains > 0)` to the end of the ready queue.

```
vector< vector<float>> RoundRobin(vector< pair<int, pair<float, float>>> processes, ofstream& outputFile){
    vector< pair<int, pair<float, float>>> original(processes.begin(), processes.end());
    sort(processes.begin(), processes.end(), IncArrTime);

    queue < pair<float, pair<float, int>>> ReadyQueue;
    vector< vector<float>> run, exec;

    int idx = 0, n = processes.size();
    float RunTime = 0, quantum = 2;

    ReadyQueue.push({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
    RunTime = max(RunTime, processes[idx].second.first);
    idx++;

    while(!ReadyQueue.empty() || idx < n){
        while(idx < n && processes[idx].second.first <= RunTime){
            ReadyQueue.push({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if (ReadyQueue.empty()) {
            RunTime = processes[idx].second.first;
            ReadyQueue.push({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        pair<float, pair<float, int>> RunningProcess = ReadyQueue.front();
        ReadyQueue.pop();

        float pID = RunningProcess.second.second;
        float arrTime = RunningProcess.second.first;
        float burstTime = RunningProcess.first;

        float usedTime = min(quantum, burstTime);
        float remTime = burstTime - usedTime;

        float waitTime = max(0.0f, RunTime - arrTime);
        float TATime = RunTime + usedTime;

        exec.push_back({pID, RunTime, usedTime, waitTime, RunTime+usedTime});

        RunTime += usedTime;

        while(idx < n && processes[idx].second.first <= RunTime){
            ReadyQueue.push({processes[idx].second.second, {processes[idx].second.first, processes[idx].first}});
            idx++;
        }

        if(remTime == 0){
            run.push_back({pID, arrTime, original[pID-1].second.second, original[pID-1].second.first, TATime});
        } else {
            ReadyQueue.push({remTime, {arrTime, pID}});
        }
    }
    outputFile << "RoundRobin (quantum = 2) :\n";
    GanttChart(exec, outputFile);
    return run;
}
```

Predicted Algorithm

The algorithms are being predicted after calculating important statistical values like mean, variance and standard deviation from the values of arrival times and burst times given in `input.txt`. Based on comparison between these indicators, we are predicting the algorithm as follows :

- If both the standard deviation of burst times (SDBT) and arrival times (SDAT) are less than their respective averages, the SJF algorithm is predicted.
- If SDBT is less than the average burst time, the SRTF algorithm is predicted.
- If more than half of the processes have burst times greater than the average burst time, the LJF algorithm is predicted.
- Otherwise, the Round Robin algorithm is predicted.

Certainly! Here's how you can compile and execute a C++ program named `cpu_scheduling` on Windows, macOS, and Linux:

Compilation and Execution

For Windows

1. **Compilation**: Open Command Prompt and run:

```
g++ -o cpu_scheduling cpu_scheduling.cpp
```

2. **Execution**:

```
cpu_scheduling.exe
```

Make sure the input file `input.txt` is in the same directory as `cpu_scheduling.exe`.

For macOS and Linux

1. **Compilation**: Open Terminal and run:

```
g++ -o cpu_scheduling cpu_scheduling.cpp
```

2. **Execution**:

```
./cpu_scheduling
```

Make sure the input file `input.txt` is in the same directory as the executable `cpu_scheduling`.

Input and Output Files

- **Input File (`input.txt`)**:
  - The input file should contain process information in the following format:
  - The process ID must start from 1 and move forward like 2, 3, 4 and so on.

    ```
    ProcessID ArrivalTime BurstTime
    ```

  - Example:

    ```
    1 15 8
    2 7 9
    3 7 2
    4 11 5
    5 18 5
    ```

- **Output File (`output.txt`)**:
  - The output file will contain the results of the scheduling algorithms including Gantt charts and average waiting and turn-around times.
  - Example:

```
Predicted Algorithm : SJF or SRTF

FCFS :
|   P2   |   P3   |   P4   |   P1   |   P5   |

7       16       18       23       31       36

Average Waiting Time : 7.4, Average Turn-around Time : 13.2



SJF :
|   P3   |   P2   |   P4   |   P5   |   P1   |

7       9       18       23       28       36

Average Waiting Time : 5.4, Average Turn-around Time : 11.2



LJF :
|   P2   |   P1   |   P4   |   P5   |   P3   |

7       16       24       29       34       36

Average Waiting Time : 10.4, Average Turn-around Time : 16.2



SRTF :
|   P3   |   P2   |   P4   |   P2   |   P5   |   P1   |

7       9       11       16       23       28       36

Average Waiting Time : 5, Average Turn-around Time : 10.8



RoundRobin (quantum = 2) :
|   P2   |   P3   |   P2   |   P4   |   P2   |   P1   |   P4   |   P2   |   P5   |   P1   |   P4   |   P2   |   P5   |   P1   |   P5

7       9       11       13       15       17       19       21       23       25       27       28       29       31       33

Average Waiting Time : 10.2, Average Turn-around Time : 16
```

By following the instructions and understanding the provided details, one can effectively utilize and analyze various CPU scheduling algorithms. This repository serves as a comprehensive guide and implementation for these algorithms.