Summary Report: CPU Scheduling Algorithms Implementation

Introduction This document provides a detailed analysis of the CPU scheduling code, which implements three different scheduling algorithms: First Come First Serve (FCFS), Round Robin (RR), and Shortest Job First (SJF). Each section breaks down the code into functional blocks, explaining their roles and execution flow.

Class Definition: Process Management

Class Name: Proccess  
Purpose: The Proccess class models a process in the system, containing attributes relevant to scheduling.

Attributes:

* id: Unique identifier for the process.
* arrival\_time: Time at which the process arrives.
* burst\_time: Total execution time required.
* waiting\_time: Time spent waiting in the queue.
* turnaround\_time: Total time from arrival to completion.
* completion\_time: When the process finishes execution.
* remaining\_time: Remaining burst time (for RR and SJF calculations).

Constructor:

* Initializes attributes and ensures unique process IDs.

First Come First Serve Algorithm

Concept:

* The simplest CPU scheduling algorithm.
* Processes are executed in order of their arrival time.

Implementation Steps:

1. Sort processes based on arrival time.
2. Iterate through processes and update completion, turnaround, and waiting times.
3. Compute averages and display results.

Round Robin Algorithm

Concept:

* We keep iterating over the queue till the queue is empty
* If a process does not complete within the time quantam, it is placed back in the queue.

Implementation Steps:

1. Initialize a queue with all processes.
2. Process execution in cycles until all processes complete.
3. Decrement remaining time based on time quantum.
4. Requeue unfinished processes.
5. Calculate turnaround and waiting times.

Hurdles:

1. I thought about making the program have a loop that increase the same amount as the time quantum and then check remaining time of the process and compare it to the time quantum, but then I had no iterator for the Processes so I would have make another loop for it.
2. Also Thought about making a vector of pairs <Process,int> in which I store the object and the remaining time, but then I would have had to change the remaining time in the vector and in the object attributes and the reordering would have been a nightmare .

Shortest Job First Algorithm

Concept:

* The process with the shortest burst time is executed first.
* Non-preemptive: Once execution starts, the process runs to completion.
* More efficient than FCFS but may cause starvation of longer jobs.

Implementation Steps:

1. Find shortest available process at the current time.
2. Execute the process and update completion, turnaround, and waiting times.
3. Repeat until all processes finish.

Hurdles:

1. I initially thought about looping for a set time (100) and checking if a new process arrived, if there is a new process we sort the vector<pair<Process, int>> that holds the object and remaining\_time again and see which process to run now, but looks inefficient and the sorting on every step would increase the time complexity.
2. I thought about making a global vector that would store the arrival times and using it to determine when each process should run. But again that would separate arrival times from the object data, making it harder to track which process corresponds to which arrival time.

Conclusion This program demonstrates three different CPU scheduling techniques:

* FCFS is simple but may lead to high waiting times.
* Round Robin ensures fairness but increases overhead.
* SJF minimizes turnaround time but may starve longer processes.

Selecting the right algorithm depends on system requirements and workload characteristics.

Source Code

#include <iostream>

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

using namespace std;

vector <int> Arrival;

// Proccess class that holds all the information about a process

class Proccess {

public:

int id;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

int completion\_time;

int remaining\_time;

Proccess(int Proccess\_id, int arr\_time, int burst\_t)

{

id = Proccess\_id;

arrival\_time = arr\_time;

burst\_time = burst\_t;

waiting\_time = 0;

turnaround\_time = 0;

completion\_time = 0;

remaining\_time = burst\_t;

Arrival.push\_back(arrival\_time);

}

};

vector<Proccess> Proccess\_List; // List to keep track of all processes

void addProcess(int id, int arrival, int burst)// Adds a new process to the list of processes while creating an object aswell

{

Proccess\_List.push\_back(Proccess(id, arrival, burst));

}

//AI

bool compareArrival(const Proccess& a, const Proccess& b) // Comparison function that returns true if process 'a' arrives earlier than process 'b'

{

return a.arrival\_time < b.arrival\_time;

}

//

void First\_Come\_First\_Serve()

{

// Sorts the process list based on arrival time in ascending order

sort(Proccess\_List.begin(), Proccess\_List.end(), compareArrival);

int current\_time = 0;

for (Proccess &Proc : Proccess\_List)

{

if (current\_time < Proc.arrival\_time) // if the current time still hasn't reached the set time for the proccess ex. time is 2, proccess starts at 3;

current\_time = Proc.arrival\_time;

Proc.completion\_time = current\_time + Proc.burst\_time;

Proc.turnaround\_time = Proc.completion\_time - Proc.arrival\_time; //Turn Around Time = Completion Time – Arrival Time

Proc.waiting\_time = Proc.turnaround\_time - Proc.burst\_time; //Waiting Time = Turn Around Time – Burst Time

current\_time += Proc.burst\_time;

}

double average\_turn = 0;

double average\_wait = 0;

cout << "First Come First Serve Algorithmn :-" << endl << endl;

for (Proccess Proc : Proccess\_List)

{

cout << "Proccess " << Proc.id << " Turn Around Time was : " << Proc.turnaround\_time << " and Waiting time was : " << Proc.waiting\_time << endl;

average\_turn += Proc.turnaround\_time;

average\_wait += Proc.waiting\_time;

}

average\_turn /= Proccess\_List.size();

average\_wait /= Proccess\_List.size();

cout << "The Average Turn Around Time was: " << average\_turn << endl;

cout << "The Average Waiting time was :: " << average\_wait << endl << endl;

}

// solved some issues with ai ( queue should be of type pointers to a class and passing by reference )

void Round\_Robin\_Algorithm(int Time\_Quantum)

{

vector<Proccess> local\_proccess\_list = Proccess\_List; // Create a local copy

queue<Proccess\*> queue;

int current\_time = 0;

// Push all processes into the queue

for (Proccess& proc : local\_proccess\_list)

{

queue.push(&proc);

}

while (!queue.empty())

{

Proccess\* Proc = queue.front();

queue.pop();

if (current\_time < Proc->arrival\_time)

{

current\_time = Proc->arrival\_time;

}

if (Proc->remaining\_time > Time\_Quantum)

{

Proc->remaining\_time -= Time\_Quantum;

current\_time += Time\_Quantum;

queue.push(Proc); // Re-add process to queue

}

else

{

current\_time += Proc->remaining\_time;

Proc->completion\_time = current\_time;

Proc->turnaround\_time = Proc->completion\_time - Proc->arrival\_time;

Proc->waiting\_time = Proc->turnaround\_time - Proc->burst\_time;

Proc->remaining\_time = 0;

}

}

double average\_turn = 0;

double average\_wait = 0;

cout << "Round Robin Algorithm :-" << endl << endl;

for (Proccess Proc : local\_proccess\_list)

{

cout << "Proccess " << Proc.id << " Turn Around Time was : "

<< Proc.turnaround\_time << " and Waiting time was : "

<< Proc.waiting\_time << endl;

average\_turn += Proc.turnaround\_time;

average\_wait += Proc.waiting\_time;

}

average\_turn /= local\_proccess\_list.size();

average\_wait /= local\_proccess\_list.size();

cout << "The Average Turn Around Time was: " << average\_turn << endl;

cout << "The Average Waiting time was :: " << average\_wait << endl << endl;

}

void Shortest\_Job\_First()

{

vector<Proccess> local\_Proccess\_List = Proccess\_List;

int current\_time = 0;

int completed = 0;

int n = local\_Proccess\_List.size();

while (completed < n)

{

Proccess\* shortest = nullptr;

// Find the process with the shortest remaining time that has arrived

for (Proccess& proc : local\_Proccess\_List)

{

if (proc.arrival\_time <= current\_time && proc.remaining\_time > 0) // Only consider processes that have arrived and are not yet completed

{

if (!shortest || proc.remaining\_time < shortest->remaining\_time) // Update if it's the first process found or has a shorter remaining time than the current shortest

shortest = &proc;

}

}

if (shortest)

{

current\_time += shortest->remaining\_time;

shortest->completion\_time = current\_time;

shortest->turnaround\_time = shortest->completion\_time - shortest->arrival\_time;

shortest->waiting\_time = shortest->turnaround\_time - shortest->burst\_time;

shortest->remaining\_time = 0;

completed++;

}

else

{

current\_time++;

}

}

double average\_turn = 0;

double average\_wait = 0;

cout << "Shortest Job First Algorithm :-" << endl << endl;

for (Proccess Proc : local\_Proccess\_List)

{

cout << "Proccess " << Proc.id << " Turn Around Time was : "

<< Proc.turnaround\_time << " and Waiting Time was : "

<< Proc.waiting\_time << endl;

average\_turn += Proc.turnaround\_time;

average\_wait += Proc.waiting\_time;

}

average\_turn /= local\_Proccess\_List.size();

average\_wait /= local\_Proccess\_List.size();

cout << "Avg Turnaround Time: " << average\_turn << endl;

cout << "Avg Waiting Time: " << average\_wait << endl << endl;

}

int main()

{

addProcess(1, 0, 5);

addProcess(2, 0, 3);

addProcess(3, 0, 8);

First\_Come\_First\_Serve();

Shortest\_Job\_First();

Round\_Robin\_Algorithm(3);

}