CPU Scheduler Visualizer

FCFS (FIRST COME FIRST SERVE)->

- Sort the Processes According to their Arrival Time

- Each process has two values:

  Arrival time (process[0]): When the process arrives in the queue.

  Burst time (process[1]): The time it needs to complete execution.

- Processes are executed in the order they arrive, without preemption.
- The CPU may be idle if a process arrives later than the current CPU time.
- The FCFS function starts by logging that FCFS scheduling is beginning.
- Processes is a **list** where each element is a list with two values: **[arrival time, burst time].**
- log is used to track the sequence of events and actions.

Variables ->

- **totalTime**: Tracks the current CPU time.
- **count**: Tracks the process number.
- **totalWait**: Accumulates the total waiting time of all processes.
- **resList**: Holds CpuProcessBar widgets that represent the execution timeline of each process.

- **Idle Time**: If the arrival time of the current process (process[0]) is greater than the current totalTime, the CPU is idle.
  The idle period is added as a gray bar in the resList, representing the idle time in the timeline.

- **Waiting Time**: If a process arrives (process[0]) before or exactly when the CPU is free (totalTime), it waits for some time.

  This waiting time is added to totalWait.

  The process's execution is shown in **orange** to indicate it waited.

  If no waiting is necessary, the bar is shown in **green**.

❖ **Execution**: After handling any waiting or idle time, the process runs for its burst time (process[1]).

The process is visualized with a CpuProcessBar, colored either green (if it didn't wait) or orange (if It did wait).

The totalTime is incremented by the burst time, simulating the passage of time.

❖ **Completion**: After all processes are executed:

The log records that FCFS scheduling has finished.

The function returns a CpuResult, which contains:

➢ The **average waiting time** (calculated as totalWait / processes.length).

➢ The **execution timeline** (resList) as a list of CpuProcessBar widgets.

FCFS CODE IN C++ For Better Understanding:

```cpp
#include <algorithm>
using namespace std;
struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival time
    int burst;      // Burst time (time required for execution)
};
// Function to perform FCFS scheduling
void fcfsScheduling(vector<Process>& processes) {
     int n = processes.size(); // Number of processes
    vector<int> waitingTime(n);  // Waiting time for each process
    vector<int> turnaroundTime(n);  // Turnaround timefor each process
    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    // Start execution at time 0
    int currentTime = 0;

    // Traverse each process
    for (int i = 0; i < n; i++) {
        Process& p = processes[i];  // Reference to the current process

        // Waiting time for the current process
        if (currentTime < p.arrival) {
            currentTime = p.arrival; // If the current time is less than the
arrival time, idle until the process arrives
        }
        waitingTime[i] = currentTime - p.arrival; // Waiting time = current time - arrival time
        currentTime += p.burst;  // Update the current time after executing the process

        // Turnaround time = burst time + waiting time
        turnaroundTime[i] = waitingTime[i] + p.burst;

        // Add to total waiting and turnaround times
        totalWaitingTime += waitingTime[i];
        totalTurnaroundTime += turnaroundTime[i];
    }

    // Display the process details
    cout << "PID\tArrival\tBurst\tWaiting\tTurnaround\n";
    for (int i = 0; i < n; i++) {
        cout << processes[i].pid << "\t"
             << processes[i].arrival << "\t"
             << processes[i].burst << "\t"
             << waitingTime[i] << "\t"
             << turnaroundTime[i] << "\n";
    }

    // Display average waiting and turnaround time
    cout << "\nAverage Waiting Time: " << (float)totalWaitingTime / n << "\n";
    cout << "Average Turnaround Time: " << (float)totalTurnaroundTime / n << "\n";
}

int main() {
    // List of processes with their arrival and burst times
    vector<Process> processes = {
        {1, 0, 5},  // Process 1: Arrival at 0, Burst time 5
        {2, 1, 3},  // Process 2: Arrival at 1, Burst time 3
        {3, 2, 8},  // Process 3: Arrival at 2, Burst time 8
        {4, 3, 6}   // Process 4: Arrival at 3, Burst time 6
    };

    // FCFS scheduling
    fcfsScheduling(processes);

    return 0;
}
```

Shortest JOB First->

- ❖ The function SJF (Shortest Job First) simulates the scheduling of processes using the SJF algorithm. In this algorithm, the process with the smallest burst time (execution time) is executed next. If a process arrives during the execution of another process and has a shorter burst time, the current process is paused, and the new process starts executing.

❖ Initialization:

You start by setting up essential variables like totalTime, count, and totalWait to keep track of the current time, number of processes handled, and the total waiting time, respectively.

A list **resList** is used to store the visual representation of each process as a bar.

colors are generated randomly for each process to distinguish between them visually.

A **delayProcess** is used as a placeholder when no processes are available.

currentProcess is initialized with the delayProcess.

currentWork keeps track of the work being done in the current time slice.

A Queue called **backlog** holds processes that are waiting to be executed.

I. 2. Main Loop Logic:

The loop continues until all processes are handled. Inside the loop, the following steps happen:

II. a. Completing a Process:

- If the currentProcess[1] (remaining time for the current process) reaches 0, the current process is considered completed.
- A visual representation of the process is added to resList with its start time, end time, and color.
- If there are processes in the backlog, the next process is popped from it and assigned to currentProcess.
- If the backlog is empty, the delayProcess (idle process) is assigned to currentProcess.

III. b. Handling New Arrivals:

- While there are still processes left (count <= processes.length - 1), it checks if the process has arrived (processes[count][0] <= totalTime).
- For each process that has arrived, it is compared with the currentProcess.
  - If the new process has a shorter burst time (processes[count][1] < currentProcess[1]), it interrupts the currentProcess.
  - The remaining time of the current process is saved, and it is added to the backlog to be executed later.

- If the new process has a longer burst time, it is added to the backlog for later execution.

IV. **c. Handling Idle Time:**

- If there are no processes to execute (currentProcess is delayProcess) and all processes have arrived, the loop ends.

V. **d. Advancing Time:**

- At each time unit (totalTime++), the remaining burst time of currentProcess is decremented (currentProcess[1]--).
- currentWork++ increments the work done in this time slice.
- Each process in the backlog waits for one time unit, so their total wait time (totalWait) increases.

VI. **3. Waiting Time Calculation:**

- Waiting time is calculated as the total time processes spend in the backlog.
- The final average waiting time is calculated as totalWait / processes.length and displayed in the result.

VII. **4. Output (Visualization):**

- Each process's execution is visually represented in the resList as a colored bar, with its start time, end time, and process ID.
- The CpuResult widget is used to display the average waiting time and the process bars.

VIII. **Key Features:**

- **Preemption:** Your implementation allows preemption. If a new process arrives and has a shorter burst time, it preempts the current running process.
- **Backlog:** Processes that are interrupted are saved in the backlog, and they will be resumed when no shorter process is available.
- **Idle Time:** The delayProcess represents idle CPU time when no processes are available, which is handled separately to ensure the correct execution order.

Code in C++ For Better understanding->

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Process {
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int startTime;
    int endTime;
    int id;
};

bool compareArrival(const Process &p1, const Process &p2) {
    return p1.arrivalTime < p2.arrivalTime;
}

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for (int i = 0; i < n; i++) {
        cout << "Enter arrival time and burst time for process " << i + 1 <<
": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].id = i + 1;
        processes[i].remainingTime = processes[i].burstTime;
    }

    // Sort by arrival time
    sort(processes.begin(), processes.end(), compareArrival);

    // Priority queue to handle the backlog of processes, prioritized by the
remaining time (Shortest Job First)
    auto cmp = [](const Process &p1, const Process &p2) {
        return p1.remainingTime > p2.remainingTime;
    };
    priority_queue<Process, vector<Process>, decltype(cmp)> backlog(cmp);

    int totalTime = 0;
    int totalWaitTime = 0;
    int count = 0;
    vector<Process> completedProcesses;

    Process currentProcess = {-1, -1, -1, -1, -1, -1}; // idle process

    while (count < n || !backlog.empty() || currentProcess.remainingTime > 0)
{
        // Handle newly arrived processes
        while (count < n && processes[count].arrivalTime <= totalTime) {
            backlog.push(processes[count]);
            count++;
        }
```

```cpp
        // If the current process is finished, store its execution
        if (currentProcess.remainingTime == 0 && currentProcess.id != -1) {
            currentProcess.endTime = totalTime;
            completedProcesses.push_back(currentProcess);
            currentProcess = {-1, -1, -1, -1, -1, -1}; // set to idle
        }

        // If there is no running process, pick the next one from the backlog
        if (currentProcess.remainingTime <= 0 && !backlog.empty()) {
            currentProcess = backlog.top();
            backlog.pop();

            // Process starts running, calculate its start time
            if (currentProcess.remainingTime == currentProcess.burstTime) {
                currentProcess.startTime = totalTime;
            }
        }

        // Process the current running process
        if (currentProcess.remainingTime > 0) {
            currentProcess.remainingTime--;
        }

        totalTime++; // increment time

        // Add waiting time for all processes in the backlog
        for (auto &p : backlog) {
            totalWaitTime++;
        }
    }

    // Calculate and display the results
    cout << "\nProcess execution order with start and end times:\n";
    for (const auto &p : completedProcesses) {
        cout << "Process " << p.id << ": Start time = " << p.startTime << ",
End time = " << p.endTime << endl;
    }

    double avgWaitTime = (double)totalWaitTime / n;
    cout << "\nAverage waiting time: " << avgWaitTime << endl;

    return 0;
}
```

IX. **Sorting Order:**

- The backlog is a **min-heap** (implemented with a priority queue) where the process with the **smallest remainingTime** comes first.
- The custom comparator (cmp) ensures that the process with the least remainingTime has the highest priority (i.e., is dequeued first).

This lambda function (cmp) compares two processes p1 and p2. It returns true if p1.remainingTime is greater than p2.remainingTime, meaning that p2 should come before p1 in the priority queue.

X. **In Summary:**

- The backlog sorts processes in **ascending order of remaining time**.
- At each time step, the process with the **shortest remaining time** is selected from the backlog for execution.

This reflects the **preemptive SJF** scheduling (also known as **Shortest Remaining Time First (SRTF)**), where the currently running process can be preempted if a newly arriving process has a shorter burst time.

RR (Round Robin) ->

In the **Round Robin (RR)** scheduling algorithm, each process is assigned a fixed **time slice (quantum)**, during which it can execute. After this time slice expires, the process is preempted and moved to the back of the **queue**. The next process in the queue is then scheduled to run. This cycle continues until all processes have completed.

Let's go through how this would work in a C++ implementation, similar to the previous SJF explanation, but with Round Robin:

Key Characteristics of Round Robin:

1. **Time Slice (Quantum)**:
   - Each process gets a **fixed amount of time** (quantum) to execute.
2. **Queue Management**:
   - Processes are stored in a **FIFO queue** (first-in-first-out), ensuring that processes get equal time shares in a cyclic manner.
3. **Preemption**:
   - If a process does not finish within its time slice, it is **preempted** and moved to the back of the queue.
4. **Arrival of New Processes**:
   - New processes are added to the queue when they arrive. If a process arrives while another process is executing, it waits in the queue.

XI. How Preemption Works in RR:

1. **Quantum Expiration**:
   - If the currently running process exhausts its quantum (let's say 4 units), it is **preempted** and moved to the back of the queue, and the next process in line is scheduled.
2. **Process Completion**:
   - If a process finishes before its quantum expires (i.e., its remaining time is less than the quantum), it is removed from the system, and the next process in the queue is scheduled.
3. **New Process Arrival**:
   - When a new process arrives, it is added to the end of the queue and will be executed once the current process either finishes its quantum or completes.

XII. Example of Round Robin Code Structure

In C++, this would typically involve managing a **queue** and a **quantum** value. Here's how it might be structured:

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

// Define a structure for processes
struct Process {
    int id;             // Process ID
    int arrivalTime;    // Arrival time of the process
    int burstTime;      // Total burst time required
    int remainingTime;  // Remaining time to finish execution
};

// Function to implement Round Robin Scheduling
void roundRobin(vector<Process> &processes, int quantum) {
    queue<Process> readyQueue;
    int currentTime = 0;
    int n = processes.size();
    int index = 0; // Track the arrival of processes

    // Add the first process to the ready queue (assuming processes are sorted
by arrival time)
    if (index < n && processes[index].arrivalTime <= currentTime) {
        readyQueue.push(processes[index]);
        index++;
    }
    while (!readyQueue.empty()) {
        Process currentProcess = readyQueue.front();
        readyQueue.pop();

        // Check if the process can finish within the quantum
        if (currentProcess.remainingTime <= quantum) {
            // Process completes within this time slice
            currentTime += currentProcess.remainingTime;
            cout << "Process " << currentProcess.id << " completed at time "
<< currentTime << endl;
            currentProcess.remainingTime = 0;  // Process is finished
        } else {
            // Process is preempted after using the quantum
            currentProcess.remainingTime -= quantum;
            currentTime += quantum;
            cout << "Process " << currentProcess.id << " executed for " <<
quantum << " time units, remaining time: " << currentProcess.remainingTime <<
endl;

            // Put it back at the end of the queue if it's not finished
            readyQueue.push(currentProcess);
        }

        // Add new processes that have arrived by currentTime
        while (index < n && processes[index].arrivalTime <= currentTime) {
            readyQueue.push(processes[index]);
            index++;
        }
    }
}

int main() {
    vector<Process> processes = {
        {1, 0, 10, 10},
        {2, 2, 5, 5},
        {3, 4, 8, 8}
    };
    int quantum = 4;

    roundRobin(processes, quantum);

    return 0;
}
```

XIII. Explanation of the Above Code:

1. **Process Structure**:
   - Each process is represented by a structure (Process), with attributes like id, arrivalTime, burstTime, and remainingTime.
2. **Queue (readyQueue)**:
   - The readyQueue manages the processes in a FIFO order. Processes that arrive are added to the queue, and the currently running process is dequeued and re-added to the end if it is not finished.
3. **Main Loop**:
   - The loop processes the queue until it's empty. For each process, it checks if the process can finish within the quantum:
     - If the process finishes, it is removed.
     - If the process cannot finish within the quantum, it is preempted (its remaining time is reduced by the quantum), and it is moved back to the end of the queue.
4. **New Process Arrival**:
   - As time progresses (currentTime), the algorithm checks for newly arriving processes and adds them to the queue.
5. **Quantum Handling**:
   - The time slice (quantum) is used to limit how long a process can execute in one go. If a process's remainingTime is larger than the quantum, it is preempted after using the quantum.

XIV. Step-by-Step Execution:

Let's assume the following processes and a quantum of 4:

Process	Arrival Time	Burst Time
P1	0	10
P2	2	5
P3	4	8

1. **At Time 0**:
   - Process P1 arrives and is executed for 4 time units (quantum expires).
   - Remaining time for P1 becomes 6 units.
   - Current time = 4.
2. **At Time 4**:

   - Process P1 is preempted and put back into the queue.
   - Process P2 arrives and is executed for 4 time units (quantum expires).
   - Remaining time for P2 becomes 1 unit.
   - Current time = 8.
3. **At Time 8**:
   - Process P2 is preempted and put back into the queue.
   - Process P3 arrives and is executed for 4 time units (quantum expires).
   - Remaining time for P3 becomes 4 units.
   - Current time = 12.
4. **At Time 12**:
   - Process P1 is executed for another 4 time units (quantum expires).
   - Remaining time for P1 becomes 2 units.
   - Current time = 16.
5. **At Time 16**:
   - Process P2 finishes execution (1 unit remaining).
   - Current time = 17.
6. **At Time 17**:
   - Process P3 is executed for its remaining 4 units and finishes execution.
   - Current time = 21.
7. **At Time 21**:
   - Process P1 finishes its remaining 2 units and completes.

XV. Key Points in RR:

- Every process gets a fixed time slice (quantum).
- Processes are executed in the order they arrive, and those that are not finished are re-added to the queue for another time slice.
- Preemption occurs when a process's time slice expires before it can finish.
- The queue ensures fairness, as every process gets equal CPU time in a cyclic manner.

TL_FCFS (Two Layer First Come First Serve)->

Two Layer Concept in TL_FCFS:

- First Layer (High Priority Queue):
    - This layer could represent processes that need urgent attention. Typically, these would be processes that either just arrived or have been preempted and are waiting for the next chance to be executed. For example, processes that have consumed their quantum but still have remaining time could be moved to the second layer.
- Second Layer (Low Priority Queue):
    - Once processes in the high priority queue (Layer 1) are done, processes from the second layer (Layer 2) are scheduled. This layer could handle processes that have already been preempted multiple times and are not as urgent.

How TL_FCFS Might Work:

1. Processes Arrival and Execution:
    - When a process arrives, it is initially placed in the first layer (high priority).
    - The processes in this first layer are executed in a First Come First Serve (FCFS) manner.
2. Time Quantum Expiry:
    - When a process is executing, it has a specific time quantum. If the quantum expires and the process is not finished, it is preempted.
    - After being preempted, the process is moved to the second layer, which handles lower priority tasks.
3. Layer 1 Drains First:
    - As long as there are processes in Layer 1, they will always be executed first. Once all Layer 1 processes are completed or moved to Layer 2, processes from Layer 2 are scheduled.
4. Queue Reordering:
    - When new processes arrive, they are added to Layer 1.
    - If there are no processes in Layer 1, processes from Layer 2 are executed in FCFS order.

In **Two Layer First Come First Serve (TL_FCFS)** scheduling, as explained earlier, we will handle two layers of queues:

- **Layer 1** will contain high-priority processes (usually new processes or those just preempted).
- **Layer 2** will handle lower-priority processes (processes that have been preempted but still need CPU time).

I'll structure the C++ code to follow this logic, using queue from the C++ STL to manage the processes in both layers.

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

struct Process {
    int id;                      // Process ID
    int arrivalTime;              // Time when process arrives
    int burstTime;                // Total CPU time required
    int remainingTime;            // Remaining CPU time for the process
};

int main() {
    // Time quantum for Round Robin scheduling
    int quantum = 4;

    // Input the number of processes
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<Process> processes(n);  // Store all processes

    // Input process information
    for (int i = 0; i < n; ++i) {
        cout << "Enter arrival time and burst time for process " << i+1 << ":
";

        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].id = i+1;
        processes[i].remainingTime = processes[i].burstTime;
    }

    // Two queues: Layer 1 (high priority) and Layer 2 (low priority)
    queue<Process> Layer1Queue;
    queue<Process> Layer2Queue;

    // Simulation variables
    int currentTime = 0;        // Keeps track of current time
    int completedProcesses = 0; // Count of completed processes
    int idx = 0;                // Index to track arriving processes

    while (completedProcesses < n) {
        // Check if new processes have arrived and add them to Layer 1
        while (idx < n && processes[idx].arrivalTime <= currentTime) {
            Layer1Queue.push(processes[idx]);
            cout << "Process " << processes[idx].id << " arrived at time " <<
currentTime << " and added to Layer 1.\n";
            idx++;
        }

        // Step 1: Execute all processes in Layer 1 first
        if (!Layer1Queue.empty()) {
            Process currentProcess = Layer1Queue.front();
            Layer1Queue.pop();

            // Execute process for the quantum
            if (currentProcess.remainingTime <= quantum) {
                currentTime += currentProcess.remainingTime;
                currentProcess.remainingTime = 0;
                completedProcesses++;
                cout << "Process " << currentProcess.id << " completed at time
" << currentTime << endl;
            } else {
                // Process can't complete, preempt it and move it to Layer 2
                currentProcess.remainingTime -= quantum;
                currentTime += quantum;
                Layer2Queue.push(currentProcess);
                cout << "Process " << currentProcess.id << " moved to Layer 2,
remaining time: " << currentProcess.remainingTime << endl;
            }
        }
        // Step 2: If Layer 1 is empty, process Layer 2
        else if (!Layer2Queue.empty()) {
            Process currentProcess = Layer2Queue.front();
            Layer2Queue.pop();

            if (currentProcess.remainingTime <= quantum) {
                currentTime += currentProcess.remainingTime;
                currentProcess.remainingTime = 0;
                completedProcesses++;
                cout << "Process " << currentProcess.id << " completed at time
" << currentTime << endl;
            } else {
                // Process can't complete, continue in Layer 2
                currentProcess.remainingTime -= quantum;
                currentTime += quantum;
                Layer2Queue.push(currentProcess);
                cout << "Process " << currentProcess.id << " executed in Layer
2, remaining time: " << currentProcess.remainingTime << endl;
            }
        }
        // If both layers are empty, advance time (idle state)
        else {
            currentTime++;
        }
    }

    cout << "All processes completed at time " << currentTime << endl;
    return 0;
}
```