**ACM Open Project - 2024**

**CPU Scheduler**

**Ojjas Madare | Computer Science and Engineering**

* Name: Ojjas Madare
* Enrolment Number: 22114064
* Email Id: [ojjas\_rm@cs.iitr.ac.in](mailto:ojjas_rm@cs.iitr.ac.in)
* Mobile Number: 9579388598

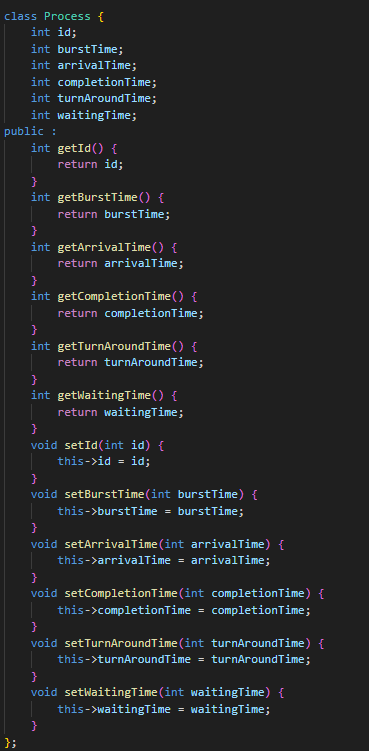
**INTRODUCTION**

A CPU scheduler is a critical component of an operating system, responsible for managing the execution of processes by determining the order in which processes access the central processing unit (CPU). The primary goal of a CPU scheduler is to optimize CPU utilization, ensuring that the CPU is kept as busy as possible by efficiently allocating time to various processes. Several scheduling algorithms are employed to achieve this, including First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Shortest Job Remaining First (SJRF). FCFS schedules processes in the order they arrive, while SJF prioritizes processes with the shortest execution time. RR allocates a fixed time slice to each process in a cyclic order, ensuring fair CPU time distribution, and SJRF, a preemptive version of SJF, interrupts the current process if a shorter job arrives, aiming to minimize the average waiting time. By implementing these algorithms, a CPU scheduler can effectively manage process execution, enhancing overall system performance and responsiveness.

***The project enhances understanding of OS process management, provides hands-on coding experience with scheduling algorithms, and improves problem-solving skills in resource allocation and system performance optimization.***

**DETAILS**

The code is entirely written in C++ and includes essential calculations for analyzing processes. These calculations include completion time (the time when the process finishes execution), turnaround time (the total time from when the process arrives until it completes execution), and waiting time (the time a process spends in the ready queue before getting the CPU for execution). To facilitate these calculations, a “Process” class has been implemented.



In the provided code snippet, the “Process” class defines several attributes:

**Attributes:**

* **id** - An integer representing a unique identifier for a process.
* **burstTime** - An integer representing the amount of time a process requires to execute.
* **arrivalTime** - An integer representing the time a process arrives in the system.
* **completionTime** - An integer representing the time a process finishes execution.
* **turnAroundTime** - An integer representing the total time a process spends in the system.
* **waitingTime** - An integer representing the time a process spends waiting for execution.

**Methods**

* **Getter methods**: These methods allow retrieving the values of the class's attributes. There is a getter method for each of the class's attributes.
* **Setter methods**: These methods allow modifying the values of the class's attributes. There is a setter method for each of the class's attributes.

**Function:** compareByArrival

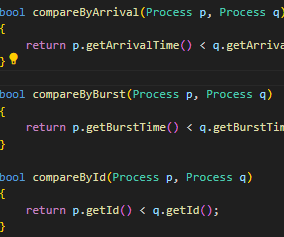
* **Compares:** Arrival time of two processes.
* **Returns:** True if the first process arrives earlier.

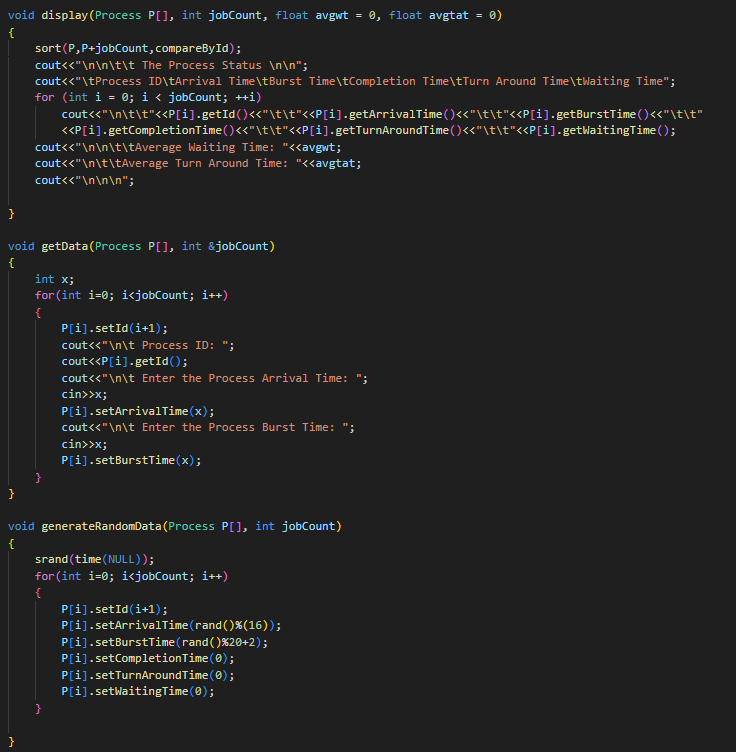
**Function:** compareByBurst

* **Compares:** Burst time (execution time) of two processes.
* **Returns:** True if the first process has a shorter burst time.

**Function:** compareById

* **Compares:** ID of two processes.
* **Returns:** True if the first process has a smaller ID.



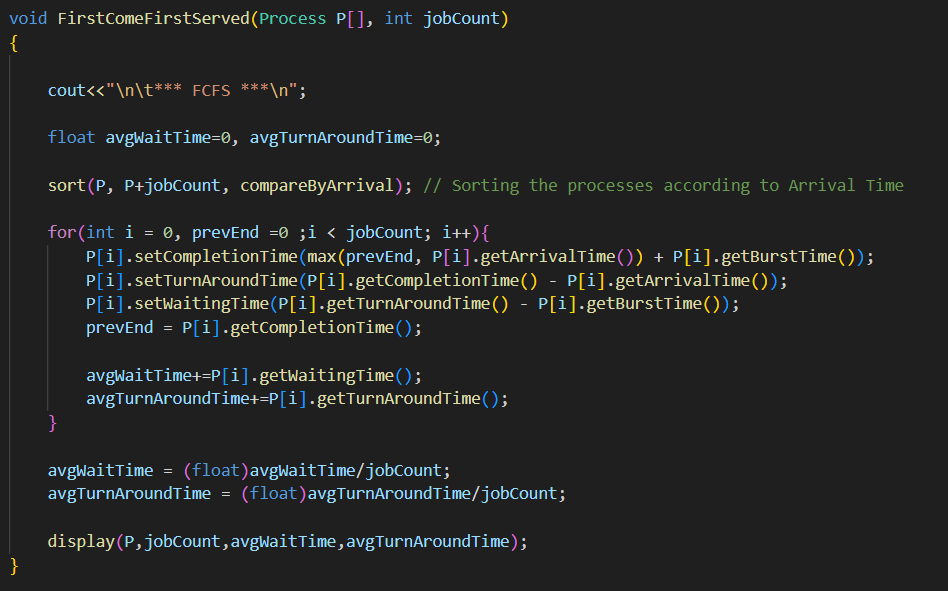


The display function takes four arguments:

* Process P[]: This is an array of Process objects.
* int jobCount: This is an integer representing the number of processes in the array.
* float avgwt: This is a float (decimal number) representing the average waiting time (likely passed in as a calculated value).
* float avgtat: This is a float representing the average turnaround time (likely passed in as a calculated value).

The function prints a table header and then iterates through the Process array, printing each process's information on a separate line.

getData & generateRandomData : takes input by using setters.



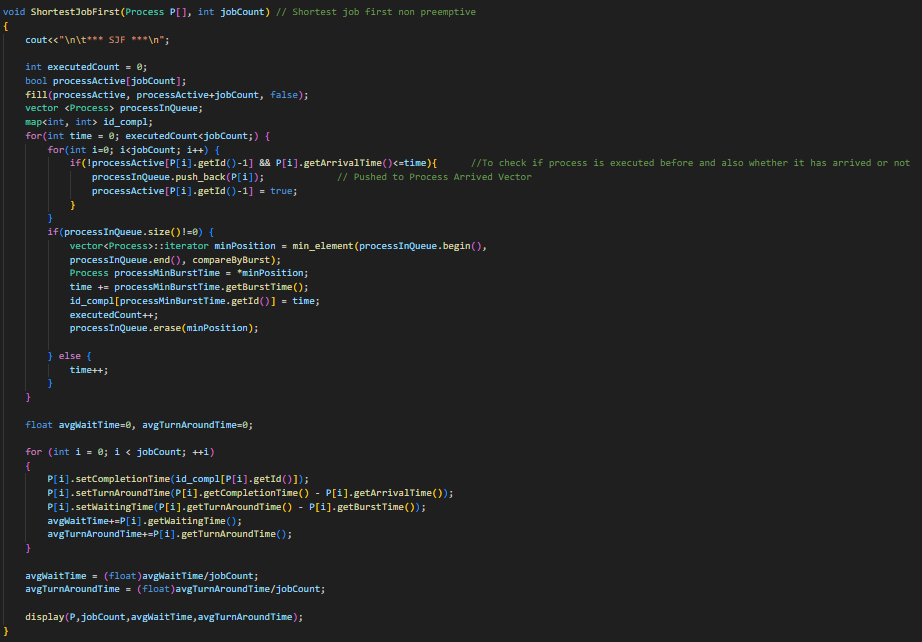
**First-Come, First-Served (FCFS) Scheduling Algorithm**

The First-Come, First-Served (FCFS) scheduling algorithm processes tasks in the order they arrive in the system. The code provided implements this algorithm, taking an array of processes and their count as input.

**Code Workflow**

1. **Initialization:** The function starts by displaying a header for the FCFS algorithm. It then initializes variables to accumulate the total waiting time and turnaround time for calculating the average values later.
2. **Sorting by Arrival Time:** To ensure the processes are executed in the order they arrive, the array of processes is sorted based on their arrival times. This sorting step is crucial for the FCFS algorithm to function correctly.
3. **Main Scheduling Loop:** The core of the algorithm iterates through the sorted list of processes. For each process:
   * The completion time is calculated. If the process arrives after the previous process has completed, it starts immediately. Otherwise, it waits until the previous process finishes.
   * The turnaround time is calculated as the difference between the completion time and the arrival time of the process.
   * The waiting time is calculated as the difference between the turnaround time and the burst time of the process.
   * The previous end time is updated to the current process's completion time for the next iteration.

Ffdfdfdf fdf



**Shortest Job First (SJF) Scheduling Algorithm**

This code snippet implements the Shortest Job First (SJF) scheduling algorithm in a non-preemptive manner. This algorithm schedules processes based on their burst time, meaning the process with the shortest burst time is executed next.

**Initialization and Setup**

The code begins by printing a header for the SJF algorithm and initializing various data structures:

* executedCount: Tracks the number of processes that have been executed.
* processActive: A boolean array to check if a process has already been added to the scheduling queue.
* processInQueue: A vector that holds processes which have arrived and are ready to be scheduled.
* id\_compl: A map that stores the completion time for each process, indexed by process ID.

**Main Scheduling Loop**

The main loop runs until all processes have been executed (executedCount < jobCount). Within this loop:

1. **Process Arrival Check:**
   * For each process, it checks if the process has arrived (P[i].getArrivalTime() <= time) and if it has not been added to the queue before (!processActive[P[i].getId() - 1]).
   * If both conditions are met, the process is added to processInQueue and marked as active.
2. **Process Selection and Execution:**
   * If there are processes in the queue (processInQueue.size() != 0), it selects the process with the shortest burst time using min\_element and the compareByBurst comparator.
   * The selected process is executed by adding its burst time to the current time (time += processMinBurstTime.getBurstTime()).
   * The completion time of the executed process is recorded in the id\_compl map.
   * The executed process is then removed from the queue, and executedCount is incremented.
3. **Idle CPU Handling:**
   * If no processes are in the queue, the time is incremented to simulate the CPU waiting for the next process to arrive.

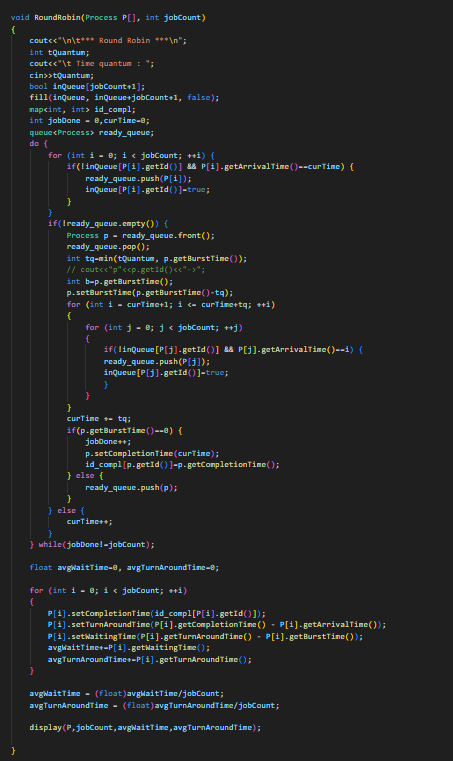


**Shortest Job Remaining First (SJRF) Scheduling Algorithm**

The SJRF algorithm is a preemptive variant of Shortest Job First (SJF) scheduling. It selects the process with the smallest remaining burst time to execute next. If a new process arrives with a smaller burst time than the currently executing process, it preempts the current process and starts executing the new one.

**Code Explanation**

1. **Initialization:**
   * The function starts by printing a header indicating the start of SJRF scheduling.
   * It initializes time to keep track of the current time, executedCount to count how many processes have finished executing, and avgTurnAroundTime, avgWaitTime to calculate average turnaround time and waiting time respectively.
   * processInQueue is a vector that holds processes ready to be scheduled based on their arrival time and remaining burst time.
   * inQueue is a boolean array to track which processes are currently in processInQueue.
   * pid\_compl is a map that stores the completion time of each process identified by its ID.
2. **Main Scheduling Loop:**
   * The main loop continues until all processes have been executed (executedCount == jobCount).
   * It iterates through each process and checks if it has arrived (P[i].getArrivalTime() <= time) and whether it is already in the queue (inQueue[i] == false).
   * If both conditions are met, the process is added to processInQueue and marked as in the queue (inQueue[i] = true).
3. **Process Execution:**
   * If there are processes in processInQueue (processInQueue.size() != 0):
     + It finds the process with the smallest remaining burst time using min\_element and compareByBurst comparator.
     + Decrements its burst time by 1 ((\*minPosition).setBurstTime((\*minPosition).getBurstTime() - 1).
     + Increments the current time (time++).
     + If the burst time of the process becomes 0 ((\*minPosition).getBurstTime() == 0):
       - Records its completion time in pid\_compl.
       - Increments executedCount.
       - Removes the process from processInQueue.
4. **Handling Idle CPU Time:**
   * If processInQueue is empty (processInQueue.size() == 0), the time is simply incremented (time++).



**Round Robin Scheduling Algorithm**

Round Robin (RR) scheduling is a preemptive scheduling algorithm where each process is assigned a fixed time slice or quantum. If a process doesn't finish within its time quantum, it's preempted and added back to the end of the ready queue.

**Code Explanation**

1. **Main Scheduling Loop:**
   * The main loop continues until all processes have been completed (jobDone == jobCount).
   * Within each iteration of the loop:
     + It checks for processes that have arrived at the current time (P[i].getArrivalTime() == curTime) and are not yet in the ready\_queue.
     + Such processes are added to the ready\_queue, and inQueue is updated to mark them as in the queue.
     + If the ready\_queue is not empty, indicating there are processes ready to execute:
       - The process at the front of the queue (ready\_queue.front()) is selected for execution.
       - The actual execution time (tq) is the minimum of the time quantum (tQuantum) and the remaining burst time of the process.
       - The current time (curTime) is incremented by tq, simulating the execution of the process for tq units of time.
       - If the process finishes executing within its time quantum (p.getBurstTime() == 0), it is marked as completed:
         * Its completion time is recorded in id\_compl.
         * jobDone is incremented to track completed processes.
       - If the process does not finish (p.getBurstTime() > 0), it is pushed back to the end of the ready\_queue for future execution.
     + If the ready\_queue is empty, indicating no processes are ready to execute at the current time, curTime is incremented to simulate idle CPU time.

**1.Initialization:**

* + The function starts by printing a header indicating the start of the Round Robin algorithm.
  + It prompts the user to input the time quantum (tQuantum) which defines the maximum amount of time a process can execute in one go.
  + Arrays and maps are initialized:
    - inQueue: A boolean array to track which processes are currently in the ready queue to prevent duplicate entries.
    - id\_compl: A map to store the completion time of each process identified by its ID.
  + Other variables like jobDone (to count completed jobs), curTime (to track the current time), and a queue<Process> (ready\_queue) to manage processes waiting to execute are initialized.

**FEATURES**

The provided C++ project implements several CPU scheduling algorithms, making it a comprehensive tool for studying and comparing different scheduling strategies. Here’s an analysis of its features:

**1. Variety of Scheduling Algorithms**: The project supports four CPU scheduling algorithms: First Come First Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Shortest Job Remaining First (SJRF). This variety allows users to explore different approaches to scheduling based on the characteristics of processes such as arrival time and burst time.

**2. Interactive User Interface**: The project features an interactive menu-driven interface where users can choose between different scheduling algorithms, input the number of processes manually or generate random data, and observe the results including average waiting time and turnaround time for each algorithm. This interactive approach enhances usability and facilitates experimentation with different scenarios.

**3. Data Handling**: The project includes functions to manually input process data or generate random data for testing purposes. This capability allows users to simulate various scenarios and evaluate how different scheduling algorithms perform under different conditions, such as varying arrival times and burst times.

**4. Performance Metrics**: Each scheduling algorithm computes and displays average waiting time and turnaround time for the set of processes it schedules. This feature enables users to quantitatively compare the efficiency of different algorithms in terms of how quickly processes are completed and how long they wait in the ready queue.

**5. Educational Value**: Beyond practical use, the project serves as an educational tool for learning and understanding CPU scheduling algorithms. By visualizing how each algorithm handles processes and computes metrics, users can gain insights into the strengths and weaknesses of each approach, deepening their understanding of operating system concepts.

In summary, this project combines practical utility with educational value, providing a platform to explore, compare, and analyze CPU scheduling algorithms in a user-friendly manner. It is well-suited for both learning purposes and practical experimentation in understanding the dynamics of process scheduling in operating systems.

**ADDITIONAL FEATURE**

To enrich the CPU scheduling project, several innovative features can be integrated to enhance its functionality and educational value. Visualizing scheduling algorithms through a graphical interface will provide users with real-time updates on processes navigating queues and executing on the CPU. This visual representation fosters a deeper understanding of algorithm behavior and system dynamics, making abstract concepts more tangible and engaging.

Moreover, enabling algorithm comparison allows users to use multiple scheduling strategies side-by-side. This feature illuminates the strengths and weaknesses of each algorithm by showcasing performance metrics such as turnaround time, waiting time, and CPU utilization. By simulating dynamic process generation and incorporating advanced metrics like response time and context switch counts, the project not only becomes a powerful learning tool but also prepares users to analyze and optimize scheduling strategies in practical scenarios. These additions ensure that the CPU scheduling project remains versatile and educational, catering to both beginners and advanced users in the field of operating systems.