

**CPU Scheduling Algorithms**



**Team Names:**

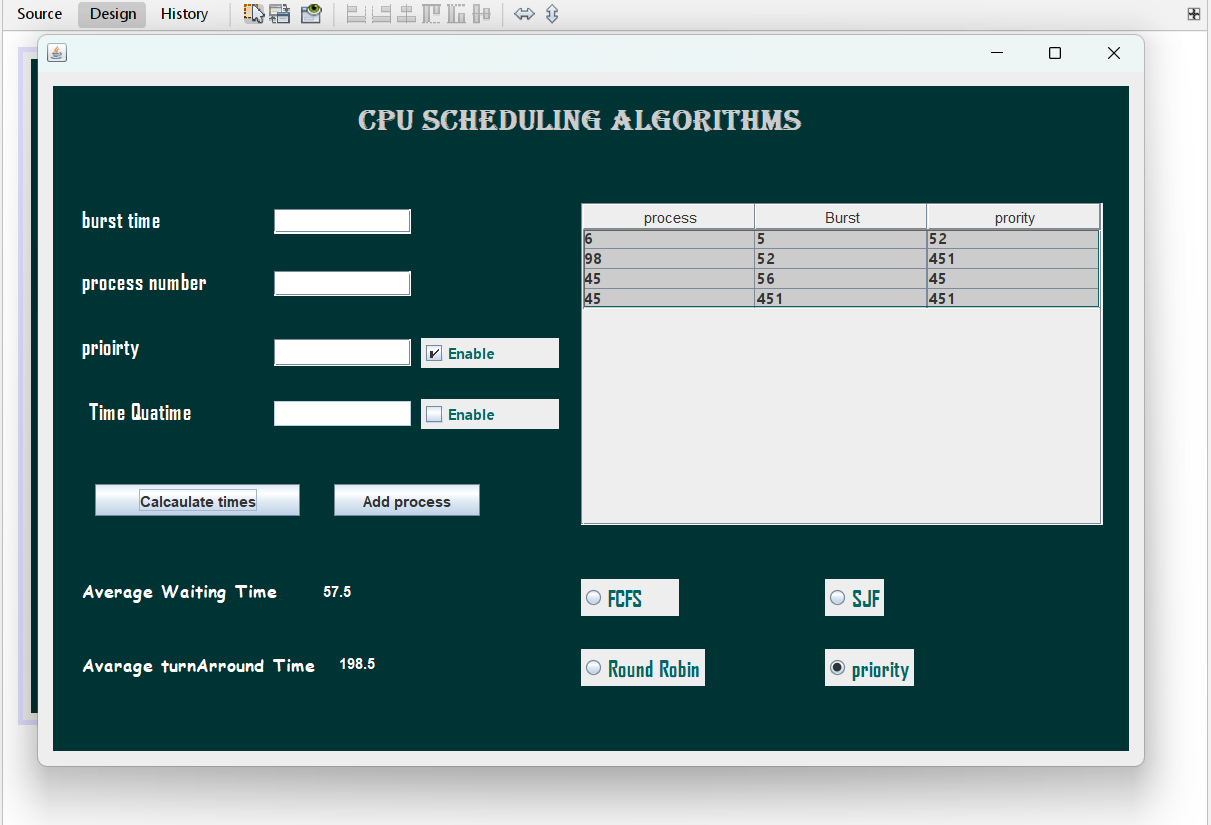
* Bahaa Mahmoud Helmy 🡪 21-00550
* Abdalla shreif Mohammed 🡪 21-00242
* Selvana Atef Kadry 🡪 21-01588
* Reham Ahmed Thabet 🡪 21-01496

**1. Introduction**

The **CPU Scheduling Algorithms Simulation Tool** is a desktop application that provides an interactive way to learn and simulate various CPU scheduling techniques. These techniques include **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Round Robin (RR)**, and **Priority Scheduling**. The system calculates key performance metrics, such as **average waiting time** and **average turnaround time**, based on user inputs.

**2. Features**

* Add processes with attributes: **Burst Time**, **Process Number**, and optionally **Priority**.
* Support for the following scheduling algorithms:
  + First Come First Serve (FCFS)
  + Shortest Job First (SJF)
  + Round Robin (RR)
  + Priority Scheduling
* Calculate and display:
  + Average Waiting Time
  + Average Turnaround Time
* GUI-based input and output visualization.



**3. Implementation Details**

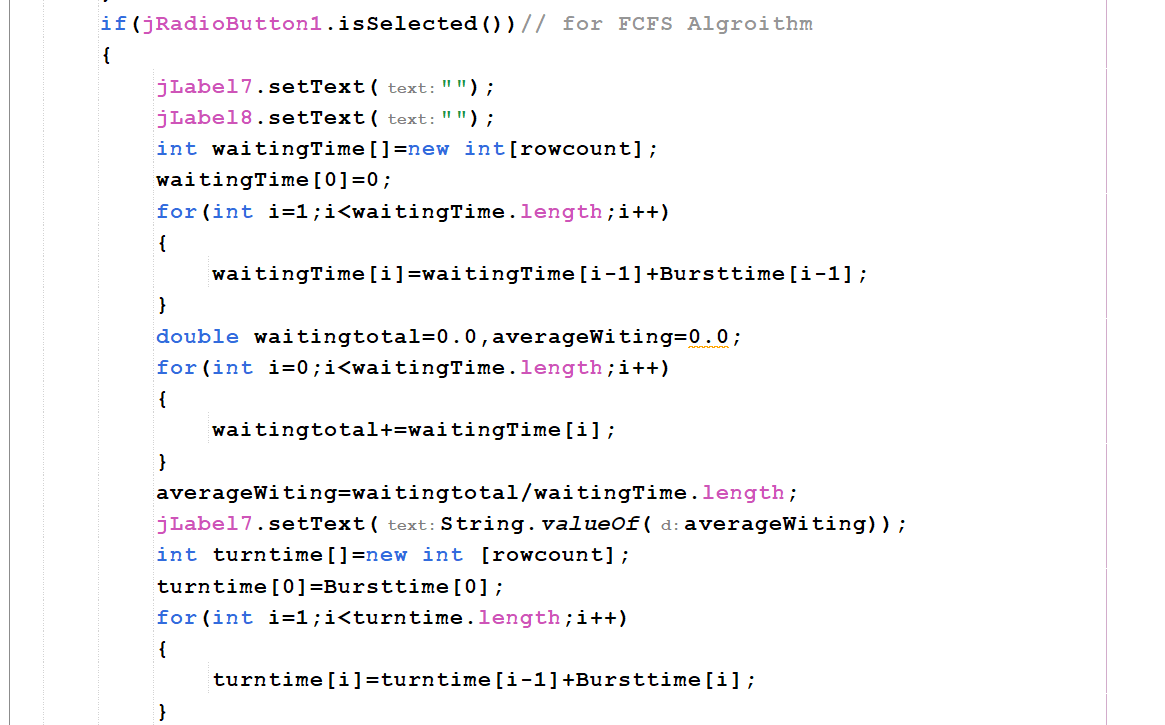
**3.1 First Come First Serve (FCFS)**

**Description**

The **First Come First Serve (FCFS)** algorithm executes processes in the order they arrive. It is a **non-preemptive** scheduling technique, which means a process will continue until it finishes before the next process starts.

**Code Implementation**

|  |
| --- |
| if (jRadioButton1.isSelected()) { // For FCFS Algorithm  jLabel7.setText("");  jLabel8.setText("");  // Waiting Time Calculation  int waitingTime[] = new int[rowcount];  waitingTime[0] = 0;  for (int i = 1; i < waitingTime.length; i++) {  waitingTime[i] = waitingTime[i - 1] + Bursttime[i - 1];  }  // Calculate Average Waiting Time  double waitingTotal = 0.0;  for (int time : waitingTime) {  waitingTotal += time;  }  double averageWaiting = waitingTotal / waitingTime.length;  jLabel7.setText(String.valueOf(averageWaiting));  // Turnaround Time Calculation  int turntime[] = new int[rowcount];  turntime[0] = Bursttime[0];  for (int i = 1; i < turntime.length; i++) {  turntime[i] = turntime[i - 1] + Bursttime[i];  }  // Calculate Average Turnaround Time  double turnTotal = 0.0;  for (int time : turntime) {  turnTotal += time;  }  double averageTurnaround = turnTotal / turntime.length;  jLabel8.setText(String.valueOf(averageTurnaround));  } |



A close up of text

Description automatically generated

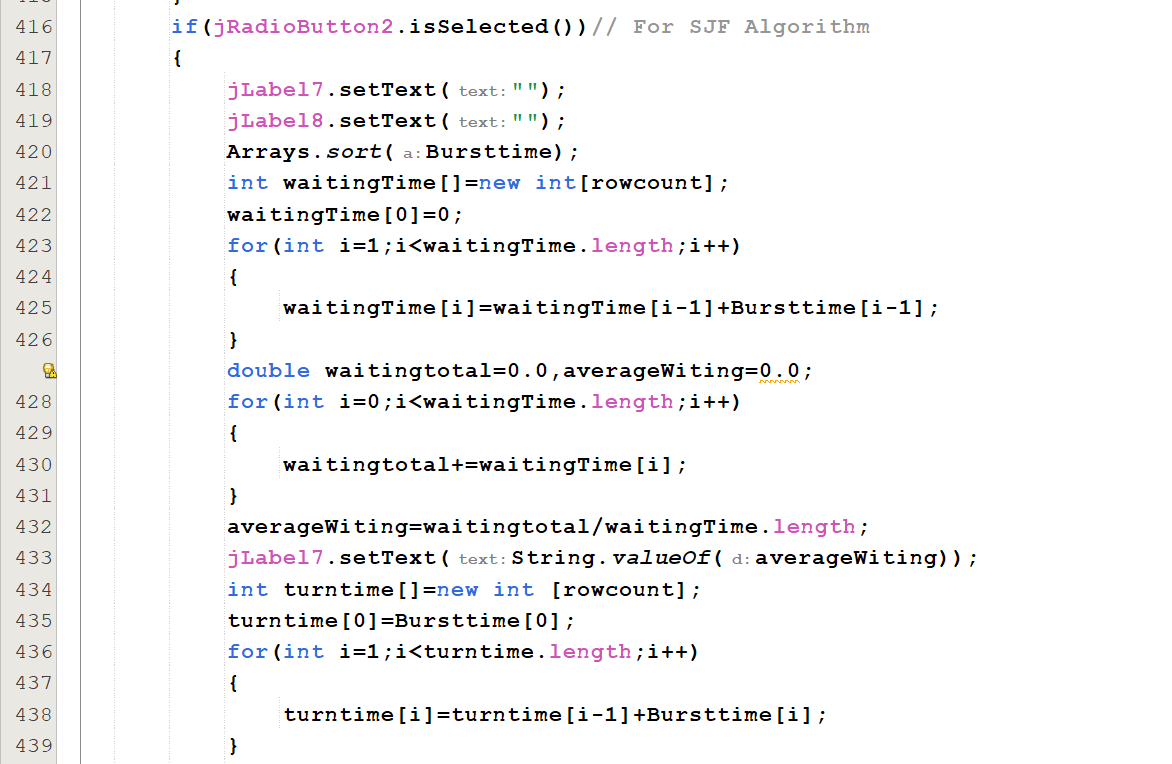
**3.2 Shortest Job First (SJF)**

**Description**

The **Shortest Job First (SJF)** algorithm executes processes in ascending order of burst time. It is a **non-preemptive** scheduling technique.

**Code Implementation**

|  |
| --- |
| if (jRadioButton2.isSelected()) { // For SJF Algorithm  jLabel7.setText("");  jLabel8.setText("");  // Sort Burst Times  Arrays.sort(Bursttime);  // Waiting Time Calculation  int waitingTime[] = new int[rowcount];  waitingTime[0] = 0;  for (int i = 1; i < waitingTime.length; i++) {  waitingTime[i] = waitingTime[i - 1] + Bursttime[i - 1];  }  // Calculate Average Waiting Time  double waitingTotal = 0.0;  for (int time : waitingTime) {  waitingTotal += time;  }  double averageWaiting = waitingTotal / waitingTime.length;  jLabel7.setText(String.valueOf(averageWaiting));  // Turnaround Time Calculation  int turntime[] = new int[rowcount];  turntime[0] = Bursttime[0];  for (int i = 1; i < turntime.length; i++) {  turntime[i] = turntime[i - 1] + Bursttime[i];  }  // Calculate Average Turnaround Time  double turnTotal = 0.0;  for (int time : turntime) {  turnTotal += time;  }  double averageTurnaround = turnTotal / turntime.length;  jLabel8.setText(String.valueOf(averageTurnaround));  } |



A computer code with text

Description automatically generated with medium confidence

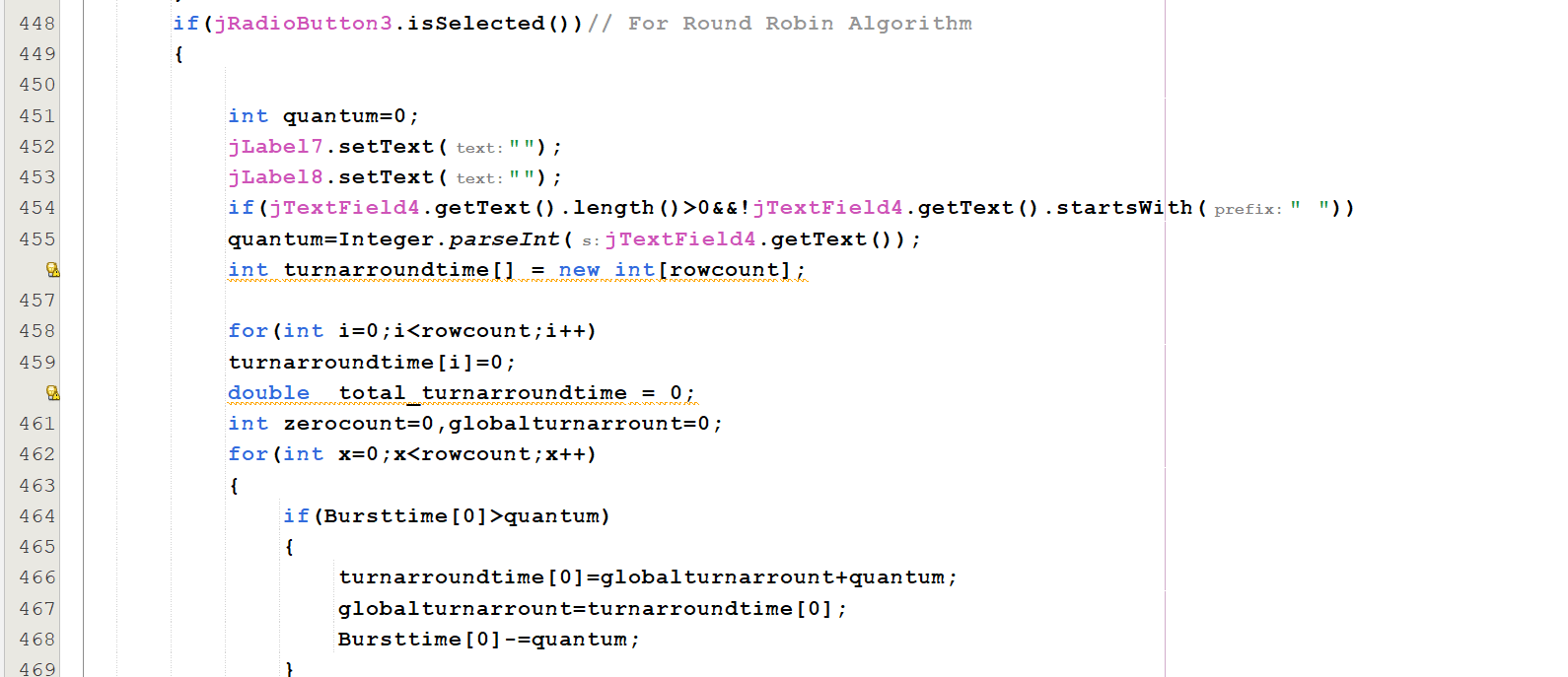
**3.3 Round Robin (RR)**

**Description**

The **Round Robin (RR)** algorithm assigns a fixed quantum time to each process. It is a **preemptive** scheduling technique where processes are cycled until completion.

**Code Implementation**

|  |
| --- |
| if (jRadioButton3.isSelected()) { // For Round Robin Algorithm  int quantum = 0;  jLabel7.setText("");  jLabel8.setText("");  if (jTextField4.getText().length() > 0 && !jTextField4.getText().startsWith(" ")) {  quantum = Integer.parseInt(jTextField4.getText());  }  int turnarroundtime[] = new int[rowcount];  for (int i = 0; i < rowcount; i++) turnarroundtime[i] = 0;  double totalTurnaroundTime = 0;  int globalTurnaround = 0;  int zeroCount = 0;  while (zeroCount < rowcount) {  zeroCount = 0;  for (int i = 0; i < rowcount; i++) {  if (Bursttime[i] > quantum) {  turnarroundtime[i] = globalTurnaround + quantum;  globalTurnaround += quantum;  Bursttime[i] -= quantum;  } else if (Bursttime[i] != 0) {  turnarroundtime[i] = globalTurnaround + Bursttime[i];  globalTurnaround += Bursttime[i];  Bursttime[i] = 0;  zeroCount++;  } else {  zeroCount++;  }  }  }  for (int time : turnarroundtime) {  totalTurnaroundTime += time;  }  double averageTurnaround = totalTurnaroundTime / rowcount;  jLabel8.setText(String.valueOf(averageTurnaround));  } |



A screenshot of a computer code

Description automatically generated

A computer code with text

Description automatically generated with medium confidence

**3.4 Priority Scheduling**

**Description**

The **Priority Scheduling** algorithm selects processes based on priority levels. It is a **non-preemptive** technique in which the process with the highest priority (lowest priority number) is executed first.

**Code Implementation**

|  |
| --- |
| if (jRadioButton4.isSelected()) { // For Priority Algorithm  jLabel7.setText("");  jLabel8.setText("");  // Sort Processes by Priority  Arrays.sort(Burstandprority, new java.util.Comparator<int[]>() {  public int compare(int[] a, int[] b) {  return Integer.compare(a[0], b[0]); // Sort by priority  }  });  // Waiting Time Calculation  int waitingTime[] = new int[rowcount];  waitingTime[0] = 0;  for (int i = 1; i < waitingTime.length; i++) {  waitingTime[i] = waitingTime[i - 1] + Burstandprority[i - 1][1];  }  // Calculate Average Waiting Time  double waitingTotal = 0.0;  for (int time : waitingTime) {  waitingTotal += time;  }  double averageWaiting = waitingTotal / waitingTime.length;  jLabel7.setText(String.valueOf(averageWaiting));  // Turnaround Time Calculation  int turntime[] = new int[rowcount];  turntime[0] = Burstandprority[0][1];  for (int i = 1; i < turntime.length; i++) {  turntime[i] = turntime[i - 1] + Burstandprority[i][1];  }  // Calculate Average Turnaround Time  double turnTotal = 0.0;  for (int time : turntime) {  turnTotal += time;  }  double averageTurnaround = turnTotal / turntime.length;  jLabel8.setText(String.valueOf(averageTurnaround));  } |



A screen shot of a computer code

Description automatically generated

**5. Summary Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Type | Preemptive | Sorting Requirement | Key Parameter |
| FCFS | Non-preemptive | No | None | Arrival Order |
| SJF | Non-preemptive | No | Burst Time | None |
| Round Robin | Preemptive | Yes | None | Quantum Time |
| Priority | Non-preemptive | No | Priority | Priority Levels |

**Conclusion**

The **CPU Scheduling Algorithms Simulation Tool** successfully demonstrates the practical implementation of four major CPU scheduling algorithms: **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Round Robin (RR)**, and **Priority Scheduling**. Through its user-friendly graphical interface, the system enables users to input processes, configure scheduling parameters, and visualize key performance metrics such as average waiting time and average turnaround time.

This tool highlights the strengths and limitations of each scheduling technique. For example:

* **FCFS** is simple but can result in longer waiting times due to the "convoy effect."
* **SJF** minimizes waiting time but may lead to starvation of longer processes.
* **Round Robin** ensures fairness but depends heavily on the quantum time chosen.
* **Priority Scheduling** efficiently handles processes with different levels of importance but can lead to lower-priority process starvation.

By providing dynamic interactions, the tool allows users to experiment with different parameters and gain a deeper understanding of how CPU scheduling algorithms impact system performance. This project is an excellent educational resource for learning about operating systems and scheduling mechanisms, bridging the gap between theoretical knowledge and practical application.