**Operating System**

**LAB PROJECT: CPU SCHEDULING**

**Department: BSCS**

**CPU SCHEDULING**

Introduction

It is the process by which the operating system decides which process gets to use the CPU at a given time. It is a fundamental concept in multitasking operating systems, where multiple processes compete for the CPU's attention. Efficient CPU scheduling ensures that system resources are used effectively and that tasks are executed in a timely manner.

**Why is CPU Scheduling Necessary?**

1. **Concurrency:**
   * Multiple processes may be running simultaneously, but the CPU can only execute one at a time. Scheduling ensures fairness and efficiency in CPU utilization.
2. **Performance:**
   * Proper scheduling minimizes waiting time, turnaround time, and response time while maximizing throughput and CPU utilization.
3. **Resource Sharing:**
   * Balances the needs of multiple processes, ensuring that high-priority tasks are completed promptly while lower-priority tasks do not starve.

**Goals of CPU Scheduling**

1. Maximize CPU Utilization
2. Maximize Throughput
3. Minimize Waiting Time
4. Minimize Turnaround Time
5. Minimize Response Time
6. Fairness

**Types of CPU Scheduling**

**1. Preemptive Scheduling:**

* A running process can be interrupted and moved back to the ready queue to allow another process to run.
* Examples: Shortest Remaining Time First (SRTF), Round-Robin (RR), Priority Scheduling (Preemptive).

**2. Non-Preemptive Scheduling:**

* Once a process starts execution, it cannot be interrupted until it completes.
* Examples: First-Come-First-Serve (FCFS), Shortest Job First (SJF), Priority Scheduling (Non-Preemptive).

**CPU Scheduling Algorithms**

1. **First-Come-First-Serve (FCFS):**
   * Processes are executed in the order of their arrival.
   * Simple but can lead to the "convoy effect" where shorter processes are delayed by longer ones.
2. **Shortest Job First (SJF):**
   * Executes processes with the shortest burst time first.
   * Minimizes average waiting time but may lead to starvation of longer processes.
3. **Priority Scheduling:**
   * Processes are executed based on priority.
   * Can be either preemptive or non-preemptive; may lead to starvation of lower-priority processes.
4. **Round-Robin (RR):**
   * Each process is assigned a fixed time slice (quantum) in a cyclic order.
   * Ensures fairness but may have higher context-switching overhead.

**Key Metrics in CPU Scheduling**

1. **Waiting Time:**
   * The time a process spends in the ready queue before execution.
2. **Turnaround Time:**
   * The total time taken for a process from submission to completion.
3. **Response Time:**
   * The time from submission of a process to the first time it starts execution.
4. **Throughput:**
   * The number of processes completed per unit of time.
5. **CPU Utilization:**
   * The percentage of time the CPU is actively executing processes.

**Practical Applications**

* **Operating Systems:**
  + Used to manage multitasking and process scheduling in systems like Windows, Linux, and macOS.
* **Real-Time Systems:**
  + Ensures time-critical tasks meet their deadlines.
* **Cloud Computing:**
  + Allocates CPU resources to virtual machines or containers efficiently.

CPU scheduling is essential for modern computing, ensuring that resources are utilized efficiently while meeting the diverse needs of different processes.

* **FCFS (First-Come-First-Serve) Implementation in the CPU Scheduler**

**First-Come-First-Serve (FCFS)** scheduling algorithm in the scheduler-CPU Windows Forms application. The FCFS scheduling algorithm executes processes in the order of their arrival, ensuring a simple and straight forward CPU scheduling mechanism.

**Code Overview**

The FCFS scheduling algorithm is implemented in the Form1 class, with buttons and controls allowing user interaction. Here's a breakdown of the key components:

**1. Initialization**

1. **Fields:**
   * int n\_fcfs: Stores the number of processes to be scheduled.
   * Queue<Process> queue\_fcfs: A queue to store processes in their order of arrival.
   * int counter\_fcfs: Tracks the number of processes added to the queue.
   * float averageWtime\_fcfs: Stores the cumulative waiting time to calculate the average.
   * int time\_fcfs: Tracks the current time in the FCFS schedule.
2. **UI Controls:**
   * flowLayoutPanel\_fcfs: Used for displaying the Gantt chart.
   * flowLayoutPanel\_fcfs\_nums: Used for displaying time stamps of the Gantt chart.
   * nOfProcesses, burstText\_fcfs, arrivalText\_fcfs: Textboxes for user input.
   * waitingText\_fcfs: Displays the calculated average waiting time.

**2. User Interaction**

1. **Step 1: Insert Number of Processes (Button1\_Click)**
   * Reads the number of processes (nOfProcesses) from user input.
   * Clears the previous Gantt chart and resets all relevant variables.
   * Enables the process insertion button (button2).
2. n\_fcfs = int.Parse(nOfProcesses.Text);
3. MessageBox.Show("ok!number of processes inserted");
4. flowLayoutPanel\_fcfs.Controls.Clear();
5. flowLayoutPanel\_fcfs\_nums.Controls.Clear();
6. button2.Enabled = true;
7. counter\_fcfs = 0;
8. averageWtime\_fcfs = 0;
9. time\_fcfs = 0;
10. **Step 2: Insert Process Details (Button2\_Click)**
    * Creates a new Process object with the given burst\_time and arrival\_time.
    * Adds the process to the queue (queue\_fcfs).
    * Updates the counter (counter\_fcfs) and displays feedback to the user.
    * Once all processes are added (counter\_fcfs == n\_fcfs), disables the button and initiates the waiting time calculation.
11. Process p\_fcfs = new Process();
12. p\_fcfs.Pid = counter\_fcfs + 1;
13. p\_fcfs.burst\_time = int.Parse(burstText\_fcfs.Text);
14. p\_fcfs.arrival\_time = int.Parse(arrivalText\_fcfs.Text);
15. queue\_fcfs.Enqueue(p\_fcfs);

**3. FCFS Scheduling Logic**

1. **Waiting Time Calculation (calculateWaitingTime\_fcfs):**
   * Dequeues processes in arrival order.
   * Updates the waiting\_time and accumulates the total waiting time (averageWtime\_fcfs).
   * Manages the Gantt chart, including process labels and timestamps.
2. tempp\_fcfs.waiting\_time = time\_fcfs - tempp\_fcfs.arrival\_time;
3. averageWtime\_fcfs += tempp\_fcfs.waiting\_time;
4. **Gantt Chart Updates:**
   * **Time Labels:** Dynamically creates Label controls for the time stamps and adds them to flowLayoutPanel\_fcfs\_nums.
   * **Process Blocks:** Dynamically creates Label controls for the processes and adds them to flowLayoutPanel\_fcfs.
5. Label tempLabel\_fcfs = new Label();
6. tempLabel\_fcfs.Enabled = true;
7. tempLabel\_fcfs.BorderStyle = BorderStyle.FixedSingle;
8. tempLabel\_fcfs.Font = new Font("Arial", 10, FontStyle.Bold);
9. tempLabel\_fcfs.Text = "P" + tempp\_fcfs.Pid.ToString();
10. **Final Calculations:**
    * Displays the average waiting time after all processes are scheduled.
    * Updates the UI with the last time label in the Gantt chart.
11. waitingText\_fcfs.Text = (averageWtime\_fcfs / n\_fcfs).ToString();

**4. Key Features**

* **Queue Implementation:** Ensures that processes are scheduled in the order of arrival.
* **Dynamic Gantt Chart:** The UI dynamically visualizes the schedule, providing a clear representation of process execution and timing.
* **Error Handling:** The algorithm checks for conditions like idle CPU time (if arrival\_time > time\_fcfs) and handles them gracefully.
* **Average Waiting Time:** Provides an easy-to-understand metric for the efficiency of the FCFS schedule.

**5. Example Workflow**

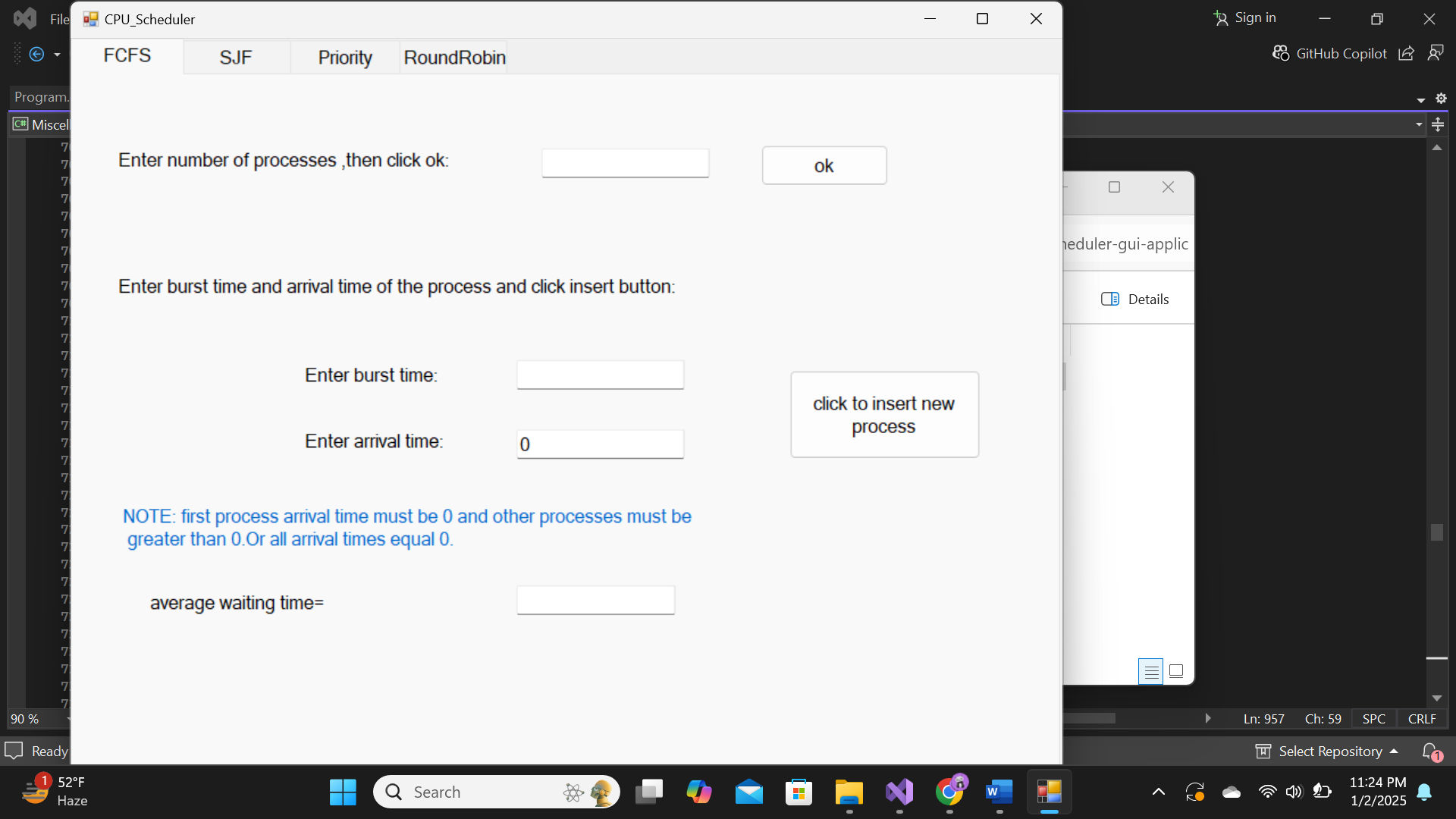
1. User inputs the number of processes (n\_fcfs).
2. User provides each process's burst\_time and arrival\_time iteratively.
3. The system schedules processes, updates the Gantt chart dynamically, and displays the average waiting time.

**6. Potential Enhancements**

* Add validation for user inputs (e.g., ensuring valid integers for burst and arrival times).
* Visual improvements for the Gantt chart (e.g., color coding or tooltips).
* Support for additional scheduling algorithms.

This implementation provides a functional and interactive simulation of the FCFS scheduling algorithm, enabling users to understand and visualize its behavior.

* **Window form Working**



* **SJF Scheduling Mode Implementation**

**Overview**

This code implements both **Preemptive** and **Non-Preemptive** Shortest Job First (SJF) scheduling algorithms in a graphical user interface (GUI) environment. It uses Gantt charts to visualize the scheduling of processes and calculates the average waiting time for processes.

**Components**

1. **No processes**
   * The Process class is used to store details of each process:
     + Pid: Process ID.
     + burst\_time: Time required for the process to execute.
     + arrival\_time: Time when the process arrives in the system (used in preemptive mode).
2. **Key Variables**
   * int num\_process: Total number of processes.
   * double sjwtime, sjaverageWtime: Variables for computing waiting time in non-preemptive SJF.
   * double prsjaverageWtime, prsjwtime: Variables for computing waiting time in preemptive SJF.
   * List<Process> vals: Stores the list of processes.
   * List<Process> premtive\_vals: Stores processes during preemptive scheduling.
   * int pd: Counter for assigning process IDs.

**GUI Controls**

1. **numericUpDown1**:
   * Used to specify the number of processes.
   * Resets state variables and enables the start button (button3).
2. **checkBox1**:
   * Toggles between Preemptive and Non-Preemptive SJF modes.
   * If checked:
     + Displays additional controls for specifying arrival time (label7, numericUpDown3).
3. **button3**:
   * Adds a process to the process list.
   * If all processes are added:
     + Schedules the processes.
     + Displays the Gantt chart and computes average waiting time.
4. **flowLayoutPanel4 and flowLayoutPanel5**:
   * Panels to display the Gantt chart.
5. **label9**:
   * Displays the average waiting time.

**Core Functions**

1. **Non-Preemptive SJF Scheduling**
   * Activated when checkBox1 is unchecked.
   * Steps:
     1. Sorts processes in ascending order of burst\_time.
     2. Iteratively computes the waiting time for each process.
     3. Visualizes the scheduling on a Gantt chart.
     4. Calculates and displays the average waiting time.
2. **Preemptive SJF Scheduling**
   * Activated when checkBox1 is checked.
   * Steps:
     1. Sorts processes in ascending order of arrival\_time.
     2. Determines the shortest available process at each time unit.
     3. Splits and reorders processes as needed for preemptive execution.
     4. Combines processes with the same Pid for Gantt chart visualization.
     5. Computes average waiting time considering process arrival and burst times.
     6. Displays the Gantt chart and average waiting time.

**Gantt Chart Generation**

* **Time Labels**: Labels added to flowLayoutPanel5 to mark execution times.
* **Process Labels**: Labels added to flowLayoutPanel4 to represent processes.
* Labels adjust their width proportionally to the burst\_time of the processes.

**Key Variables and Calculations**

1. **Average Waiting Time**:
   * **Non-Preemptive**: Average Waiting Time=Total Waiting TimeNumber of Processes\text{Average Waiting Time} = \frac{\text{Total Waiting Time}}{\text{Number of Processes}} Waiting time accumulates as processes are executed sequentially.
   * **Preemptive**: Waiting time is calculated dynamically as processes preempt each other based on arrival time and remaining burst time.
2. **Dynamic Sorting**:
   * In Non-Preemptive mode, processes are sorted once by burst\_time.
   * In Preemptive mode, processes are sorted multiple times dynamically during execution.

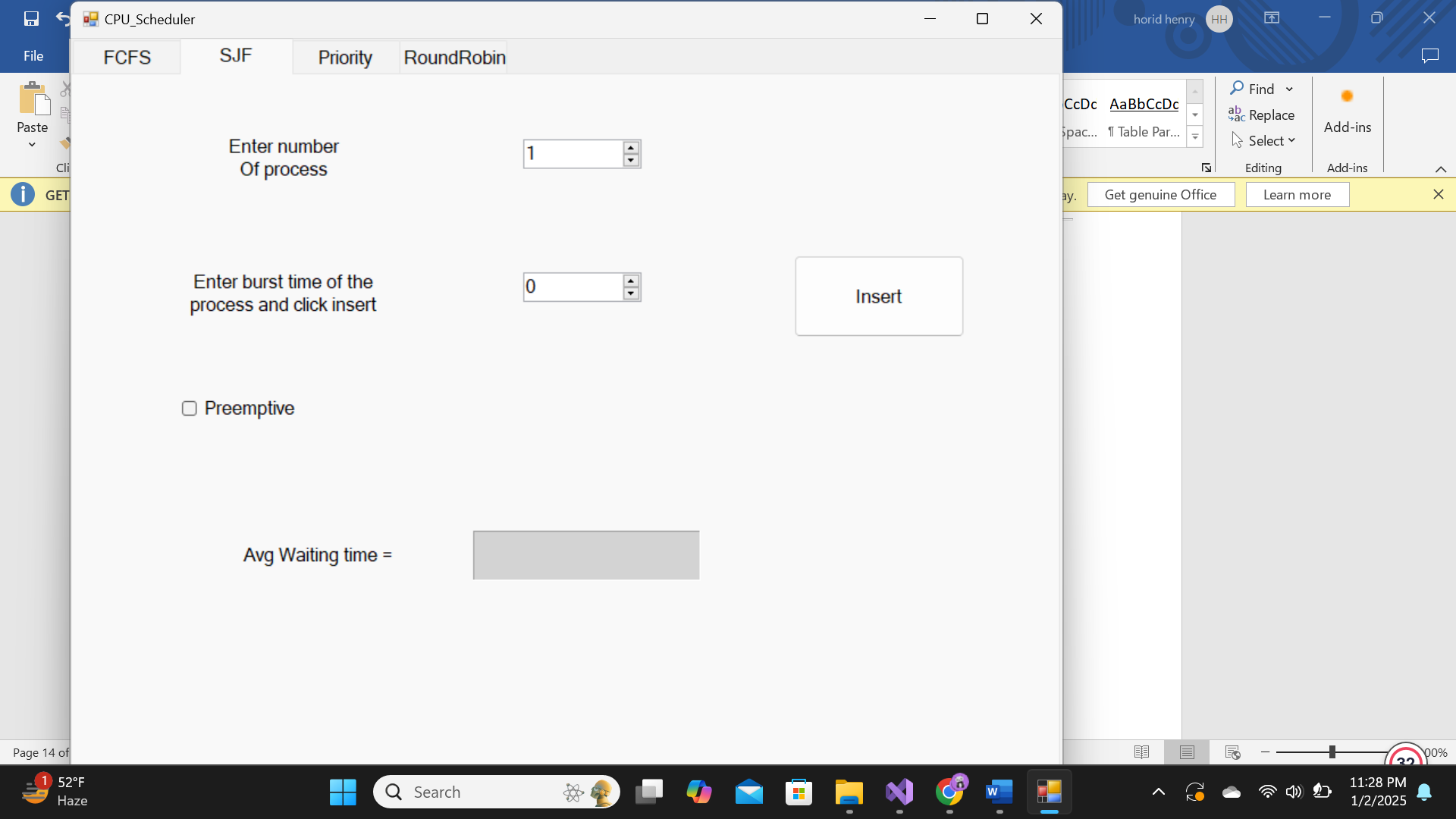
**Code Snippets**

1. **Sorting Processes (Non-Preemptive)**:
2. vals = vals.OrderBy(arr => arr.burst\_time).ToList();
3. **Preemptive Scheduling Logic**:
   * Splits the execution into smaller chunks:
   * Process pre\_sjf = new Process();
   * pre\_sjf.burst\_time = vals[i + 1].arrival\_time - vals[i].arrival\_time;
   * pre\_sjf.Pid = vals[indx].Pid;
   * pre\_sjf.arrival\_time = vals[indx].arrival\_time;
   * premtive\_vals.Add(pre\_sjf);
   * Combines processes with the same Pid:
   * if (premtive\_vals[jj].Pid == premtive\_vals[jj + 1].Pid)
   * {
   * premtive\_vals[jj + 1].burst\_time += premtive\_vals[jj].burst\_time;
   * premtive\_vals.RemoveAt(jj);
   * jj = -1;
   * }

**Enhancements & Future Work**

1. Add validation for user input to handle edge cases like 0 or negative burst/arrival times.
2. Enhance Gantt chart display for better visualization (e.g., color coding processes).
3. Include additional metrics like Turnaround Time for more comprehensive analysis.
4. Optimize the preemptive scheduling algorithm for larger datasets.

* **Window form Working**



* **Priority Scheduling**

This section of the code implements **Priority Scheduling**, including both **Preemptive** and **Non-Preemptive** modes. Priority scheduling assigns execution priority to processes based on their priority values, where lower priority numbers indicate higher priority

**Priority Scheduling Algorithm** in CPU scheduling assigns priority levels to processes, where the process with the highest priority (usually the lowest priority number) is selected for execution first. It can operate in two modes:

1. **Non-Preemptive**: A running process cannot be interrupted, even if a higher-priority process arrives.
2. **Preemptive**: The CPU switches to a higher-priority process as soon as it arrives, preempting the current process.

The algorithm ensures important tasks are handled promptly but can lead to **starvation** of low-priority processes, which can be mitigated using **aging** (gradually increasing the priority of waiting processes).

**Key Variables**

1. **Process Control Variables**
   * num\_process\_prio: Number of processes for scheduling.
   * pri\_vals: List to store non-preemptive processes.
   * premt\_pri\_vals: List to store preemptive processes.
2. **Time Calculations**
   * prwtime: Accumulated waiting time for non-preemptive scheduling.
   * praverageWtime: Average waiting time for non-preemptive scheduling.
   * prtprwtime: Accumulated waiting time for preemptive scheduling.
   * prtpraverageWtime: Average waiting time for preemptive scheduling.
3. **Other Variables**
   * pr\_pd: Process ID counter.

**UI Control Handlers**

**1. numericUpDown5\_ValueChanged**

* Triggered when the user sets the number of processes.
* Resets the scheduler and prepares the UI for a new scheduling session.
  + Clears Gantt chart containers (flowLayoutPanel7 and flowLayoutPanel8).
  + Resets accumulated waiting times and process counters.
  + Enables the process insertion button (button4).

**2. checkBox2\_CheckedChanged**

* Toggles between **Preemptive** and **Non-Preemptive** modes based on the checkbox state.
  + Displays or hides the **Arrival Time** input (label14 and numericUpDown6).

**3. button4\_Click**

* Handles process insertion and executes the scheduling logic when all processes are input.

**Scheduling Logic**

**1. Process Insertion**

* Captures process attributes:
  + burst\_time: Time required by the process.
  + priority: Priority of the process.
  + arrival\_time: Time the process arrives (if preemptive mode is active).
* Adds the process to the pri\_vals list.
* Displays a message indicating progress.

**2. Non-Preemptive Priority Scheduling**

* Triggered when all processes are inserted, and the Preemptive checkbox is unchecked.

1. **Sorting**
   * Processes are sorted by priority in ascending order.
2. **Gantt Chart**
   * For each process:
     + Calculates and displays start time, end time, and process labels on the Gantt chart.
   * Updates accumulated and average waiting times.
3. **UI Updates**
   * Displays the calculated average waiting time.
   * Resets the process list for the next session.

**3. Preemptive Priority Scheduling**

* Triggered when all processes are inserted, and the Preemptive checkbox is checked.

1. **Sorting**
   * Processes are initially sorted by arrival time.
2. **Preemptive Scheduling Logic**
   * Processes are split and scheduled based on arrival and remaining burst times.
   * Higher-priority processes preempt lower-priority ones.
3. **Merge and Simplify**
   * Combines segments of processes with the same PID for clarity.
4. **Gantt Chart**
   * Similar to non-preemptive, with the addition of tracking preemptions.
5. **UI Updates**
   * Displays the calculated average waiting time.
   * Resets process lists for future sessions.

**Key Methods**

**Process Sorting**

* Non-Preemptive: pri\_vals = pri\_vals.OrderBy(arr => arr.priority).ToList();
* Preemptive: Combines sorting by arrival time and priority dynamically during scheduling.

**Gantt Chart Updates**

* Time labels (flowLayoutPanel8) and process blocks (flowLayoutPanel7) are dynamically created and added based on calculated start and burst times.

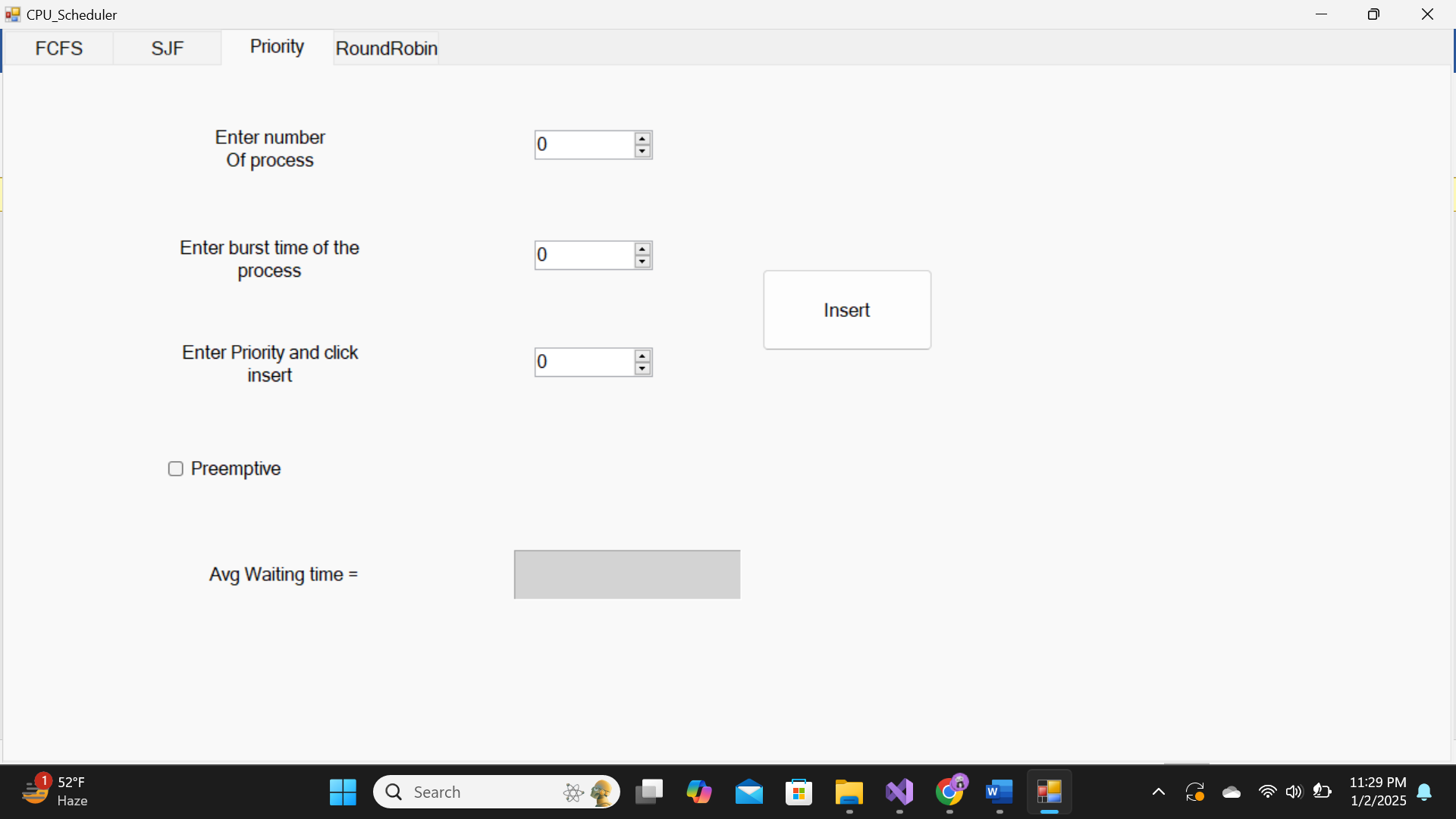
**Output**

1. **Gantt Chart**:
   * Displays process execution order with time labels.
2. **Average Waiting Time**:
   * Non-Preemptive: praverageWtime / num\_process\_prio
   * Preemptive: prtpraverageWtime / num\_process\_prio

**Preemptive vs Non-Preemptive**

| **Feature** | **Non-Preemptive** | **Preemptive** |
| --- | --- | --- |
| **Priority Change** | Fixed during execution. | Dynamically adjusts based on arrival and priority. |
| **Gantt Chart** | Contiguous execution blocks. | Split execution blocks with potential preemptions. |

* **Window form Working**



* **Round Robin Mode**

The **Round Robin (RR) Scheduling** implementation in this code is designed to simulate and calculate the CPU scheduling process based on the Round Robin algorithm. It divides the CPU time into fixed-length time slices (quantum) and cycles through all processes in a fair, time-sharing manner. Round Robin (RR) scheduling algorithm allows for the simulation and visualization of process scheduling in a GUI. The algorithm uses a time-sharing mechanism where each process is assigned a fixed time slice, or quantum, and cycles through processes in a circular order.

**Key Components and Features**

1. **Process Input**
   * **Number of Processes (n\_RR)**: Specifies the total number of processes.
   * **Quantum Time (quantum)**: The fixed time slice allocated to each process.
   * Input fields accept process-specific attributes:
     + Burst Time (burstText\_RR)
     + Arrival Time (arrivalText\_RR)
2. **GUI Elements**
   * **FlowLayoutPanels**:
     + flowLayoutPanel\_RR: Displays the process execution order in a Gantt chart.
     + flowLayoutPanel\_RR\_nums: Displays timeline labels corresponding to the Gantt chart.
   * **Text Boxes**:
     + textBox2: Number of processes input.
     + textBox4: Quantum time input.
   * **Labels**:
     + Used for Gantt chart visualization and timeline markers.
   * **Buttons**:
     + insertButton\_RR: Allows for entering process details.
     + button6\_Click\_1: Initializes the scheduler with the number of processes and quantum.
3. **Data Structures**
   * **Queue** (queue): Used to maintain a circular list of processes to be executed in RR order.
   * **Process Class**:
     + Attributes: Pid, burst\_time, waiting\_time, arrival\_time, last\_active.

**Round Robin Execution Flow**

1. **Initialization**
   * The number of processes (n\_RR) and quantum time (quantum) are set via the GUI.
   * Reset variables:
     + counter\_RR: Tracks the number of processes inserted.
     + averageWtime\_RR: Accumulates total waiting time for averaging.
     + time\_RR: Tracks elapsed time.
2. **Process Insertion**
   * Each process is entered into the queue with attributes:
     + Process ID (Pid)
     + Burst Time
     + Arrival Time
   * Once all processes are added, insertButton\_RR is disabled, and the waiting time calculation begins.
3. **Waiting Time Calculation**
   * **Process Execution**:
     + If the process arrival time is greater than the current time, it is re-enqueued.
     + For burst times shorter than the quantum, the process completes execution, and its waiting time is calculated.
     + For burst times longer than the quantum, the process executes for the quantum duration, and the remaining burst time is re-enqueued.
   * **Waiting Time Update**:
     + Calculated as the difference between the current time and the process's last active time or arrival time.
   * **Time Progression**:
     + Time (time\_RR) is updated as processes execute, and labels are added to visualize the execution timeline.
4. **Gantt Chart Generation**
   * **Visualization**:
     + A label is created for each process execution segment, showing the process ID and duration.
     + Timeline labels mark the start and end times for each process execution.
5. **Final Results**
   * The average waiting time is calculated and displayed in waitingText\_RR.

**Advantages**

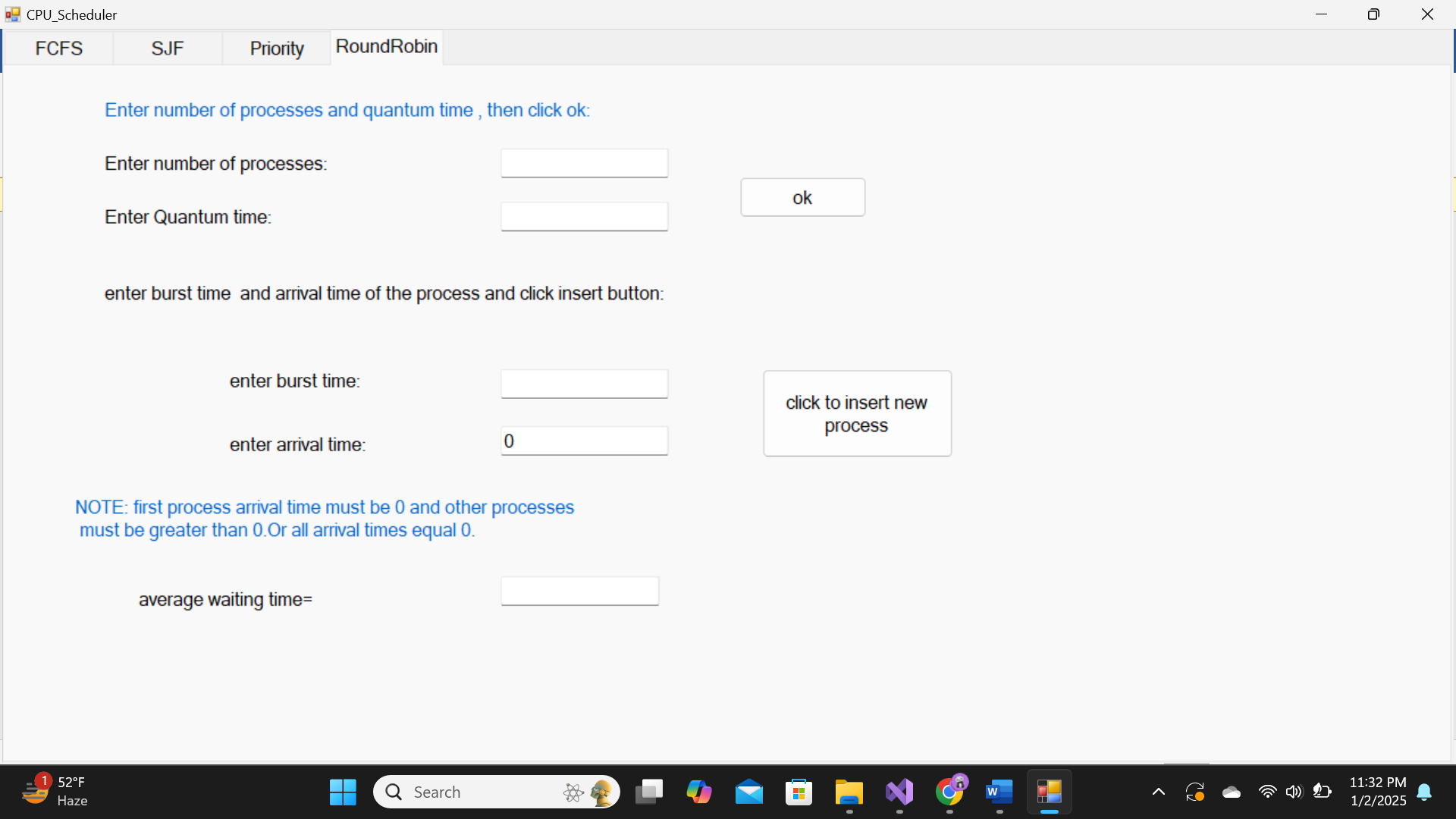
* **Fairness**: Each process gets an equal opportunity to execute.
* **Efficiency**: Effective for time-sharing systems with minimal starvation.
* **Visualization**: Gantt chart representation provides clarity on process scheduling.

**Limitations**

* **Overhead**: Frequent context switches can reduce CPU efficiency.
* **Quantum Dependency**: Performance is highly dependent on the choice of quantum.
* **Core Logic**:
  + Ensures fair and time-shared execution.
  + Handles process arrival and burst time effectively.
  + Dynamically updates the Gantt chart based on the execution.

This implementation provides a clear and visual way to understand and analyze the Round Robin scheduling algorithm.

* **Window form Working**



* **Conclusion: CPU Scheduling Algorithms**

1. **Priority Scheduling**:
   * Schedules processes based on priority; suitable for real-time systems.
   * Preemptive mode requires careful handling of arrival and burst times.
   * Can cause starvation if priorities aren't managed dynamically.
2. **Round Robin Scheduling**:
   * Ensures fairness with fixed quantum time; ideal for time-sharing systems.
   * Efficiently handles processes using a queue but can have context-switch overhead.
   * Quantum selection impacts system responsiveness and efficiency.
3. **FCFS (First-Come-First-Served)**:
   * Simple and easy to implement; processes are executed in the order of arrival.
   * Can lead to long waiting times for processes arriving later, especially if a long process arrives first.
   * Not ideal for systems with varying burst times, as it can cause inefficiency.
4. **SJF (Shortest Job First)**:
   * Prioritizes processes with the shortest burst times, leading to minimized waiting times.
   * Ideal for batch systems where burst times are known in advance.
   * Can result in starvation for longer processes and is difficult to implement in dynamic environments where burst times aren't known ahead of time.