The Superior University

📌 Project Title

FCFS Task Scheduler with Gantt Chart Visualization using Python.

👥 Group Members

Team members with roll numbers:

- Eman Jamshed (SU92-BSSEM-F23-009)
- Rukhsar Arshad (SU92-BSSEM-F23-293)
- Maryam Rashid (SU92-BSSEM-F23-025)
- Muhammad Abdullah Awan (SU92-BSSEM-F23-010)

📂 GitHub Repository

GitHub Repository Link of All Group Members with their names:

https://github.com/rukhokodar/FCFS-Task-Scheduler-with-Gantt-Chart-Visualization-Python-.git (Rukhsar Arshad).
https://github.com/Eman-code960/FCFS-Task-Scheduler-with-Gantt-Chart-Visualization-Python-.git (Eman Jamshed).
https://github.com/miriumkodar/FCFS-Task-Scheduler-with-Gantt-Chart-Visualization-Python-.git (Maryam Rashid).
https://github.com/abdullahawan001/FCFS-Task-Scheduler-with-Gantt-Chart-Visualization-Python-.git (Abdullah Awan).

🛠 Scheduling Algorithm Implemented

✅ Tick the scheduling algorithm your group implemented:

- [x] FCFS (First Come First Serve)
- [ ] SJF (Shortest Job First – Non-Preemptive)
- [ ] SJF (Preemptive)
- [ ] Round Robin

📄 Project Description

What problem your project solves

The project simulates a CPU task scheduler that follows the First Come First Serve (FCFS) algorithm. It helps visualize how processes are executed in the order they arrive, which is a fundamental concept in CPU scheduling.

In simple words, this means that tasks (or processes) are executed **in the same order** in which they are entered. The first task to arrive will be the first one to get the CPU, just like people waiting in line — first come, first served.

This project helps us understand **how an operating system decides which task to run next** when many tasks are waiting to be processed.

What inputs are required?

- **Task Name:** A label for the task (e.g., T1, Task, etc.)
- **Burst Time:** The amount of time the task needs to execute

Note: Arrival time and time quantum are not required in FCFS. The algorithm assumes tasks arrive one after another in the order they are entered.

What outputs are generated?

- It calculates:
- **Waiting Time** – how long each task has to wait before it starts.
- **Turnaround Time** – the total time from when the task was entered until it finishes.
- **Start and End Times** – when each task begins and ends.
- **Average Waiting** Time and Turnaround Time
- **Gantt Chart** displaying process execution visually

How the algorithm is implemented

When the user enters tasks (with names and burst times), the program **stores them in a list** in the exact order they are entered — this is important because **FCFS runs tasks in the same order they arrive**.

Then the scheduler goes through each task **one by one**:

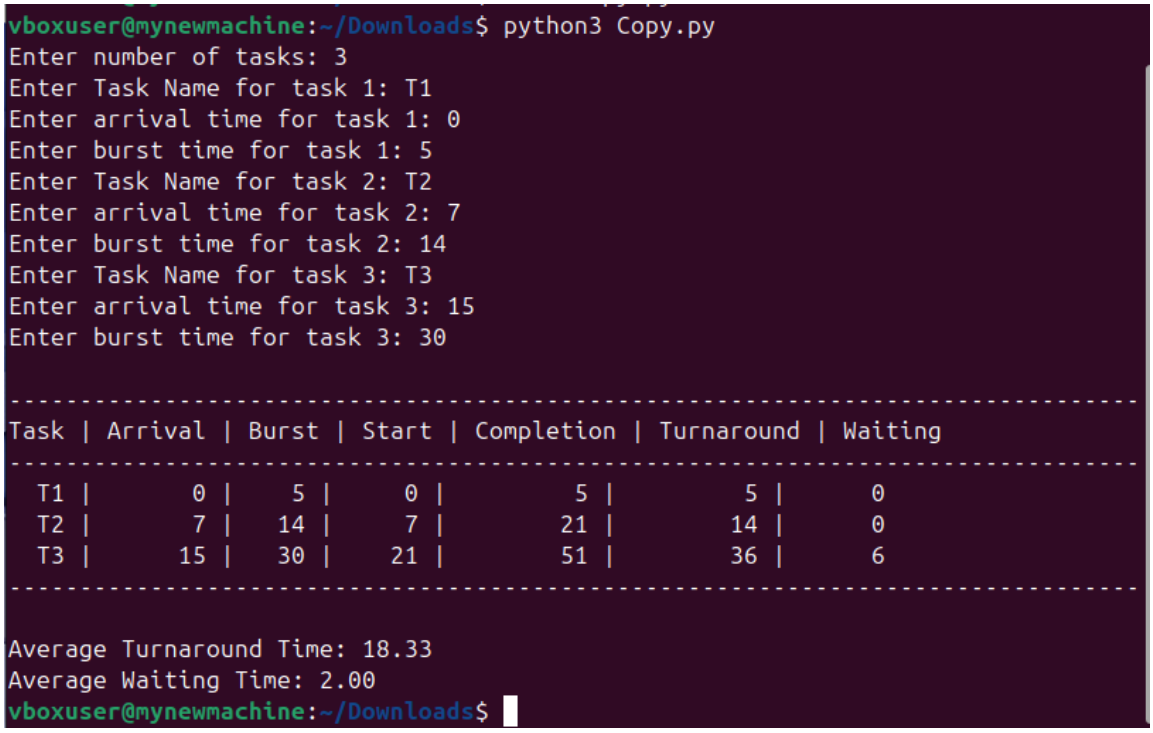
1. It sets the **start time** of each task based on when the previous task finished.
2. It calculates the **waiting time**, which is how long the task had to wait before starting.
3. It calculates the **turnaround time**, which is the total time from the start of the program until the task is completed.
4. It also records the **end time** of the task.

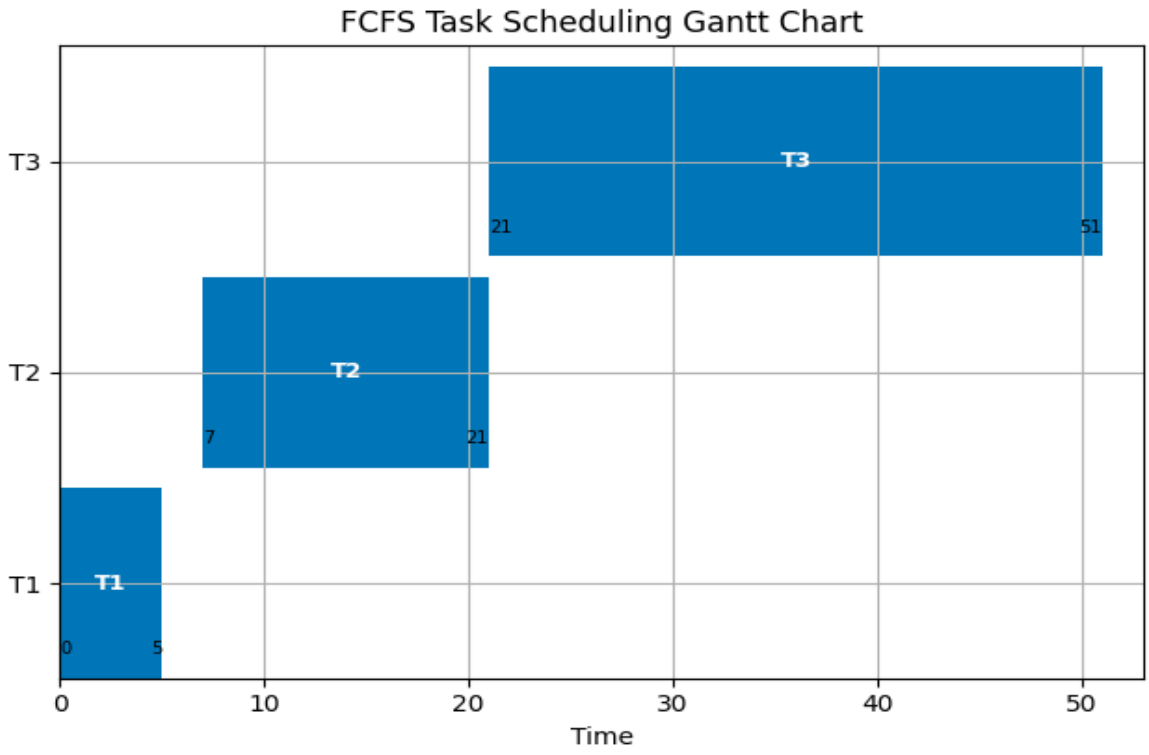
After all the tasks are scheduled, the program shows the results in a **neatly formatted table** — showing each task's burst time, waiting time, turnaround time, start and end time.

Finally, it creates a **Gantt chart** using the matplotlib library. This chart is a **visual timeline** that shows how long each task runs and in what order. It uses different colors to make each task easy to see.

Output Screenshots

1st sample Output with Gantt Chart



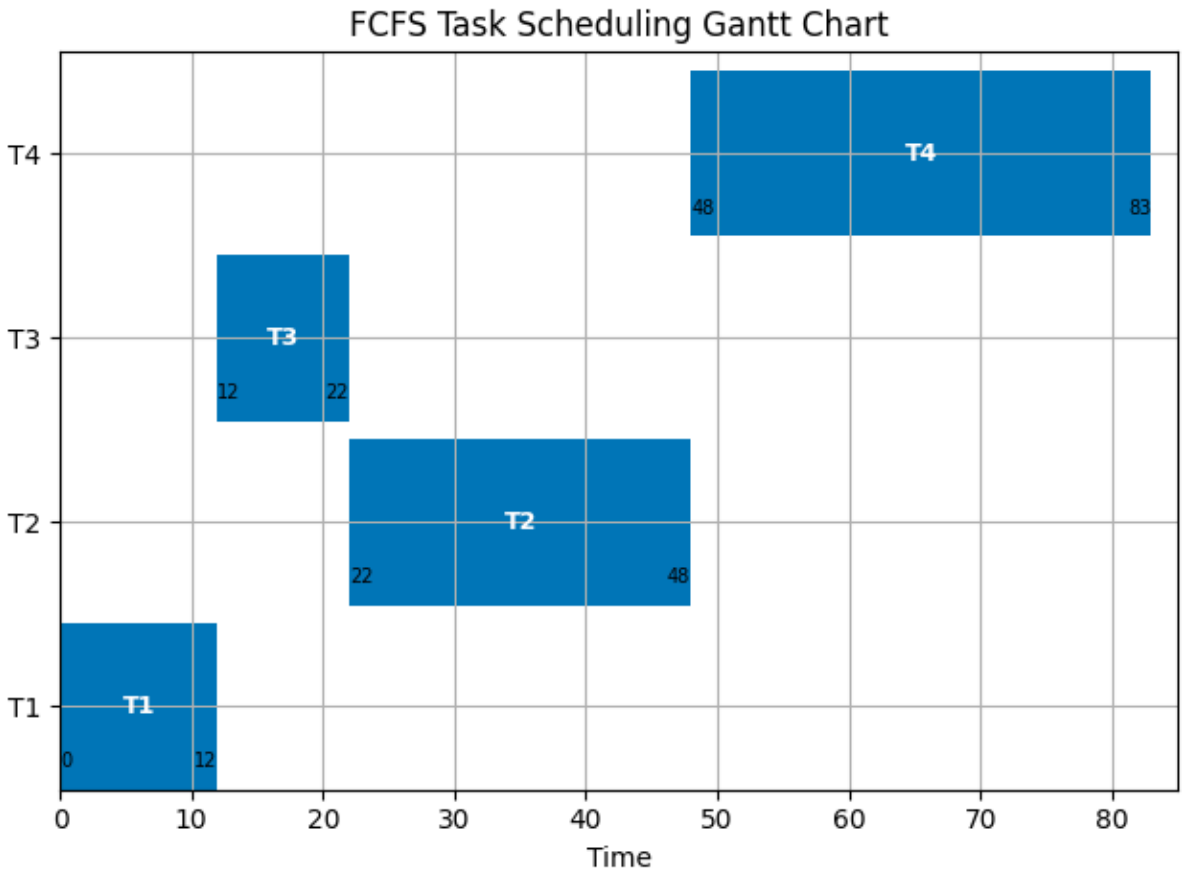


2nd sample Output with Gantt chart

```
vboxuser@mynewmachine:~/Downloads$ python3 Copy.py
Enter number of tasks: 4
Enter Task Name for task 1: T1
Enter arrival time for task 1: 0
Enter burst time for task 1: 12
Enter Task Name for task 2: T2
Enter arrival time for task 2: 13
Enter burst time for task 2: 26
Enter Task Name for task 3: T3
Enter arrival time for task 3: 8
Enter burst time for task 3: 10
Enter Task Name for task 4: T4
Enter arrival time for task 4: 28
Enter burst time for task 4: 35

------------------------------------------------------------------------------
Task | Arrival | Burst | Start | Completion | Turnaround | Waiting
------------------------------------------------------------------------------
  T1 |       0 |   12 |     0 |         12 |         12 |       0
  T2 |      13 |   26 |    22 |         48 |         35 |       9
  T3 |       8 |   10 |    12 |         22 |         14 |       4
  T4 |      28 |   35 |    48 |         83 |         55 |      20
------------------------------------------------------------------------------

Average Turnaround Time: 29.00
Average Waiting Time: 8.25
```



Code Structure & Explanation

a. Functions or Logic Used

1. Task Representation (Dictionary-Based)

Instead of using a class, each task is represented using a dictionary with the following attributes:

- **name:** Task identifier (e.g., T1, T2).
- **arrival_time:** Time when the task enters the system.
- **burst_time:** CPU time required to complete the task.
- **start_time:** Time the task begins execution.
- **completion_time:** When the task finishes.
- **turnaround_time:** Total time from arrival to completion.
- **waiting_time:** Total time the task spends waiting in the queue.

2. Scheduling Logic

The core logic follows First-Come, First-Served (FCFS) rules:

- Tasks are sorted based on arrival time.
- If the CPU is idle before a task arrives, it waits until the arrival.
- Each task's start_time, completion_time, turnaround_time, and waiting_time are calculated in sequence.

3. Output and Visualization

- The code prints a well-formatted task table showing all relevant times.

It also calculates and displays:

- **Average Turnaround Time**
- **Average Waiting Time**

4. Gantt Chart Drawing

> Uses **matplotlib. pyplot** to draw a Gantt chart.

Tasks are displayed as horizontal bars using **ax.broken_barh**.

> Each task has:

- A bar with start_time and burst_time.
- Task name label.
- Start and completion times marked.
- The Y-axis dynamically adjusts based on task count.

b. Core Logic of the Scheduling Algorithm

This code implements the FCFS (First-Come, First-Served) algorithm as follows:

1. User Input:

> Users enter the number of tasks, then for each task:

- Task name.
- Arrival time.
- Burst time.

2. Scheduling:

Tasks are sorted by arrival time.

> For each task:

- If the CPU is idle, the current time jumps to the task's arrival.
- start_time = current time.
- completion_time = start_time + burst_time.
- turnaround_time = completion_time - arrival_time.
- waiting_time = start_time - arrival_time.
-

3. Output:

> Displays all tasks in a table showing:

- Arrival, Burst, Start, Completion, Turnaround, and Waiting Time.
- Computes and displays average waiting and turnaround times.

4. Gantt Chart:

- Visual representation of task execution timeline.
- Each task shown with start and end times on a time-axis.

c. External Libraries Used

1. matplotlib. pylon

- Used to generate a Gantt chart.
- Displays each task as a bar along a timeline.
- Adds task name, start, and completion times for clarity.

2. No use of random

Unlike earlier implementations with random colors, this version uses the fixed color 'tab:blue' for all tasks to maintain simplicity and readability.

Performance Metrics

Metric	Value (Calculated After Input)
Average Waiting Time	Depends on task burst times and order
Average Turnaround Time	Depends on task burst times and order
Time Quantum	Not used — FCFS is not Round Robin

Challenges Faced

1. Incorrect Waiting Time Calculation

Challenge:

Initially, we misunderstood how to calculate waiting time in FCFS. We did not consider that each task's waiting time depends on the total burst time of all the preceding tasks. This led to inaccurate waiting time and turnaround time results.

Resolution:

After reviewing the FCFS logic, we updated the schedule () method so that each task's waiting time is correctly set to the end time of the previous task. This fixed the inconsistency and ensured correct timing values for all tasks.

2. Gantt Chart Visualization Issues

Challenge:

While implementing the Gantt chart, we faced issues with task bars overlapping or not aligning correctly along the timeline. This made the visualization unclear and confusing.

Resolution:

We used the matplotlib library to generate the Gantt chart and adjusted the Y-axis placement, limits (xlim, ylim), and text alignment. We also assigned each task a unique color for clarity. These changes resulted in a clean and accurate visual timeline.

3. Handling User Input for Task Names and Burst Times

Challenge:

User input errors, such as entering non-numeric burst times, initially caused the program to crash. Since our program depends on real-time user input, handling such cases was crucial.

Resolution:

We added input validation in the main() function using a try-except block. If the user enters invalid input, they are prompted to re-enter the data. This makes the program more robust and user-friendly.

Summary

- **Waiting Time Calculation**: We fixed incorrect waiting time calculation by ensuring that the waiting time for each task was calculated based on the finish time of the previous task.
- **Gantt Chart Visualization**: Adjusted the task placements and added unique colors to make the Gantt chart visually accurate and readable.