This section focuses on how the program will handle input data from the user and file, and how it will display the required output, based on the project requirements and the source code.

Inputs

The inputs for the process scheduling simulation consist of two main components: the process data and the user's selection of the scheduling algorithm. Here's how they are handled:

1. Process Data from a File
Source: The program reads process data from a text file named `processes.txt`.
Format: Each line in the file represents a process with the following fields, separated by one or more whitespace characters:
PID (Process ID): An integer identifying the process.
Arrival_Time: An integer representing when the process enters the system.
Burst_Time: An integer indicating the CPU time required by the process.
Example File Content:

```
PID Arrival_Time Burst_Time
1   0           5
2   2           3
3   4           7
```

Handling:
- The program assumes the first line is a header (e.g., column names) and skips it.
- It reads subsequent lines, parses the values into integers, and stores them in a `List<ProcessData>` using the `ProcessData` class.
- Error handling is implemented to catch file-not-found or malformed data exceptions, displaying an error message if issues occur.
- Implementation: The `readProcessData` method in the main `Scheduler.java` class performs this task:

java:
```
public static List<ProcessData> readProcessData(String fileName) {
    List<ProcessData> processDatas = new ArrayList<>();
    try {
        File file = new File(fileName);
        Scanner scanner = new Scanner(file);
        if (scanner.hasNextLine()) scanner.nextLine(); // Skip header
        while (scanner.hasNextLine()) {
            String[] data = scanner.nextLine().split("\\s+");
            int PID = Integer.parseInt(data[0]);
```

```
            int Arrival_Time = Integer.parseInt(data[1]);
            int Burst_Time = Integer.parseInt(data[2]);
            processDatas.add(new ProcessData(PID, Arrival_Time, Burst_Time));
        }
        scanner.close();
    } catch (Exception e) {
        System.out.println("Error reading file: " + e.getMessage());
        return null;
    }
    return processDatas;
}
```

Note:The source code (FIFO.java and SJF.java) already reads data in this format, so this approach ensures compatibility.

2. Scheduling Algorithm Selection
Source: User input via a menu-driven interface.
Options: The user selects from available scheduling algorithms (at least FIFO and SJF, as implemented in the provided code).
Format: A numeric choice entered via the console:
- 1 for FIFO (First-Come, First-Served)
- 2 for SJF (Shortest Job First)
Example Interaction:

```
Select Scheduling Algorithm:
1. FIFO
2. SJF
Enter choice: 1
```

Handling:
- The program displays a menu and uses a `Scanner` to read the user's integer input.
- Based on the choice, it calls the corresponding scheduling method (e.g., `FIFO.schedule` or `SJF.schedule`).
- Implementation: This is managed in the `main` method of `Scheduler.java`:

java:

```
public static void main(String[] args) {
    System.out.println("Select Scheduling Algorithm:");
    System.out.println("1. FIFO");
    System.out.println("2. SJF");
    System.out.print("Enter choice: ");
    Scanner scanner = new Scanner(System.in);
    int choice = scanner.nextInt();
    String fileName = "processes.txt";
    List<ProcessData> processDatas = readProcessData(fileName);
```

```
        if (processDatas == null) {
            return; // Error reading file
        }
        if (choice == 1) {
            FIFO.schedule(processDatas);
        } else if (choice == 2) {
            SJF.schedule(processDatas);
        } else {
            System.out.println("Invalid choice.");
        }
    }
```

Note: The project allows for either command-line arguments or a menu-driven approach. A menu is chosen here for simplicity and user-friendliness, but it could be adapted to use command-line arguments (e.g., `java Scheduler --algorithm FIFO`).

Outputs:

The outputs are generated after simulating the selected scheduling algorithm and include a Gantt chart, process details, and average times, as required by the project. These are displayed in the console in a clear, formatted manner.

1. Gantt Chart
Purpose: Shows the execution order of processes over time.
Format: A text-based representation with:
- Process IDs (e.g., P1, P2) between vertical bars (`|`) indicating execution periods.
- Time intervals below, showing when each process starts and ends.
Example:

```
Gantt Chart:
| P1 | P2 | P3 |
0    5    8   15
```

- P1 runs from 0 to 5, P2 from 5 to 8, P3 from 8 to 15.
Handling:
- During simulation, a `List<GanttEntry>` tracks each process's start and end times using the `GanttEntry` class:

java:
```
    public class GanttEntry {
        public ProcessData process;
        public int startTime;
        public int endTime;
```

```
        public GanttEntry(ProcessData process, int startTime, int endTime) {
            this.process = process;
            this.startTime = startTime;
            this.endTime = endTime;
        }
    }
```

- After simulation, the chart is printed by iterating over this list.

Implementation (FIFO Example):

java:

```
List<GanttEntry> ganttChart = new ArrayList<>();
int currentTime = 0;
for (ProcessData pd : processDatas) {
    if (currentTime < pd.Arrival_Time) {
        currentTime = pd.Arrival_Time;
    }
    int startTime = currentTime;
    int endTime = currentTime + pd.Burst_Time;
    ganttChart.add(new GanttEntry(pd, startTime, endTime));
    pd.Completion_Time = endTime;
    currentTime = endTime;
}
System.out.println("\nGantt Chart:");
System.out.print("|");
for (GanttEntry entry : ganttChart) {
    System.out.printf(" P%d |", entry.process.PID);
}
System.out.println();
System.out.print(ganttChart.get(0).startTime);
for (GanttEntry entry : ganttChart) {
    System.out.printf("%5d", entry.endTime);
}
System.out.println();
```

Note: The original SJF.java attempts a Gantt chart but lacks proper time intervals. This enhanced version corrects that and is applied consistently to both FIFO and SJF.

2. Process Details Table

Purpose: Displays calculated metrics for each process.

Format: A tabular output with columns:

PID: Process ID.

Arrival: Arrival Time.

Burst: Burst Time.

Completion: Completion Time (when the process finishes).
Waiting: Waiting Time (TAT - Burst Time).
Turnaround: Turnaround Time (Completion Time - Arrival Time).
Example:

```
PID Arrival Burst Completion Waiting Turnaround
 1     0    5        5       0        5
 2     2    3        8       3        6
 3     4    7       15       4       11
```

Handling:
- Processes are sorted by PID for consistency and readability.
- Metrics are calculated during simulation and stored in the `ProcessData` objects.
- The table is printed with formatted spacing.

Implementation (FIFO Example):

java

```
processDatas.sort(Comparator.comparingInt(p -> p.PID));
System.out.println("\nPID Arrival Burst Completion Waiting Turnaround");
for (ProcessData pd : processDatas) {
    System.out.printf("%3d %6d %5d %10d %7d %10d\n",
        pd.PID, pd.Arrival_Time, pd.Burst_Time,
        pd.Completion_Time, pd.Waiting_Time, pd.Turnaround_Time);
}
```

3. Average Waiting Time and Turnaround Time

Purpose: Summarizes the performance of the scheduling algorithm.
Format: Two lines displaying averages with two decimal places.
Example:

```
Average Waiting Time: 2.33
Average Turnaround Time: 7.33
```

Handling:
- Total Waiting Time and Turnaround Time are accumulated during simulation.
- Averages are calculated by dividing totals by the number of processes.

Implementation (FIFO Example):

java

```
int totalWT = 0, totalTAT = 0;
for (ProcessData pd : processDatas) {
    pd.Turnaround_Time = pd.Completion_Time - pd.Arrival_Time;
    pd.Waiting_Time = pd.Turnaround_Time - pd.Burst_Time;
```

```
    totalWT += pd.Waiting_Time;
    totalTAT += pd.Turnaround_Time;
  }
  int n = processDatas.size();
  System.out.printf("\nAverage Waiting Time: %.2f\n", (double) totalWT / n);
  System.out.printf("Average Turnaround Time: %.2f\n", (double) totalTAT / n);
```

Integration with Existing Source Code

(FIFO.java and SJF.java) provides the core scheduling logic, which I've adapted into static `schedule` methods within their respective classes. Here's how the inputs and outputs integrate:

ProcessData Class:Unified into a single definition with fields for PID, Arrival_Time, Burst_Time, Completion_Time, Waiting_Time, Turnaround_Time, and a `completed` boolean (used by SJF):

java
```
  public class ProcessData {
    public int PID, Arrival_Time, Burst_Time, Completion_Time, Waiting_Time,
Turnaround_Time;
    public boolean completed = false;
    public ProcessData(int PID, int Arrival_Time, int Burst_Time) {
      this.PID = PID;
      this.Arrival_Time = Arrival_Time;
      this.Burst_Time = Burst_Time;
    }
  }
```

FIFO.schedule: Adapted to include Gantt chart generation and sorted process details.
SJF.schedule: Enhanced to improve Gantt chart accuracy with start/end times.
Scheduler.java: Acts as the entry point, handling input collection and delegating to the chosen algorithm.

Summary

Inputs:
- Process data from `processes.txt` (PID, Arrival Time, Burst Time), with the first line skipped as a header.
- User selection of scheduling algorithm (FIFO or SJF) via a menu.

Outputs:

- A text-based Gantt chart showing process execution order and time intervals.
- A table of process details (PID, Arrival, Burst, Completion, Waiting, Turnaround), sorted by PID.
- Average Waiting Time and Turnaround Time, calculated and displayed with two decimal places.

This setup ensures that your part of the project—handling inputs and outputs—works seamlessly with your friend's scheduling algorithms, meeting all project requirements for file handling, user interaction, and output display.