COMP2432 Operating Systems Group Project

Steel-making Production Line Scheduler (PLS)

Jyotsna Venkatesan 22108825D

Tahmin Anower 22107453D

Tanya Budhrani 22097189d

## **Abstract**

This document outlines the details for a project in the COMP2432 course for the academic year 2023-2024. ⁤⁤The goal of this project is to create a Production Line Scheduling (PLS) machine specifically designed for a medium-sized steel manufacturer who owns three plants. ⁤The main objective of this system is to enhance production planning and maximize the utilization of the three plants in order to effectively schedule the various orders inputted into the system.

⁤The system achieves its purpose by accurately handling input data, effectively scheduling production tasks, generating clear reports, and presenting relevant information to the users. ⁤⁤It offers user-friendly operations based on commands, allowing users to easily add scheduling periods, generate comprehensive reports, and perform other vital tasks. ⁤⁤By implementing this PLS machine, the project aims to improve operational efficiency and profitability in the steel industry. ⁤

## **Introduction**

This project is integral for a medium-sized steel manufacturer with three production facilities, each specializing in different types of steel products: finished steel rails, semi-finished rods and wires, and crude pig iron. Despite the significant production capacity of these plants, the company recently faced a drop in profits due to inefficient production scheduling. The inability to complete several orders on time not only hurt immediate profits but also threatened the company's reputation over the long term.

The purpose of this project is to create a tool that improves how production is scheduled across the three plants. Using the principles of scheduling algorithms learned in our course at PolyU, this initiative aims to maximize the operational efficiency of each facility. The software will intelligently schedule production tasks to ensure deadlines are met and will also determine which orders should be accepted or rejected based on the plants' capacities and scheduling limitations. This approach will prevent the acceptance of unfeasible orders, thus avoiding losses and bolstering both the short-term profitability and long-term viability of the company. The project seeks to transform a current weakness into a strength, positioning the company for greater competitive success.

## **Scope/Related work**

1. Process Scheduling Algorithms: First-Come-First-Serve (FCFS):

Processes are produced in the order in which they arrive. This is done by processing orders in the sequence they are received.

1. Process Scheduling Algorithms: Shortest Job First (SJF):

This algorithm picks the process that has the smallest processing time. It is simulated by choosing orders that can be completed the quickest based on plant capacity.

1. Forks:

Forks are used in the program to create a new child process that runs concurrently with the parent process.The program can delegate specific tasks such as running a scheduling algorithm or scheduling on a day-to-day basis to the child process. This allows the parent process to manage tasks like user input and output separately from the algorithm execution, making the program more organized and manageable.

1. Pipes: Pipes are used in our code to establish a communication channel between the parent process and the child process. This allows the parent process to write the algorithm name to the pipe using the write() system call. The written data is then read by the child process from the pipe using the read() system call and stored in the algoBuffer variable.

## **Concept**

1. Shortest Job First:  
   The SJF (Shortest Job First) algorithm implemented in the code assigns orders to plants based on the concept of minimizing processing time. It starts by identifying orders that can be completed by the plants and stores the eligible orders in an array. Then, for each simulated day, the algorithm iterates through available plants and orders to find the combination with the shortest processing time. The algorithm calculates the daily production capacity of the plant and assigns the corresponding production quantity to the order. This process continues until no more allocations are possible for the day.
2. First-Come-First-Serve:

The FCFS (First-Come, First-Served) algorithm assigns orders to plants based on the order in which they appear in the list. After identifying the orders that can be completed by the plants, the algorithm simulates a fixed number of days. For each day, it iterates through the available plants and orders, starting from the beginning of the list. The algorithm assigns the daily production capacity of the plant to the order, updates the plant's statistics, and marks the plant as used for the day. This process continues until all orders have been processed or no more allocations are possible for the day.

## **Software structure of your system**

Below are the functions that we utilized in creating our PLS software along with their descriptions:

**printOrderHistory()**

This function prints the order history stored in the system. It traverses the linked list of orders and prints each order's details, such as order ID, due date, quantity, and product name, in a formatted manner. If there are no orders in the history, it displays a message indicating the absence of orders

**rejectOrder(Order\* order):**

The rejectOrder function adds an order to the rejected list by assigning it to the next available index in the global rejectedOrders array and increments the global rejectedCount variable. It also prints a rejection message indicating the order number and its due date, informing that the order cannot be completed within the specified due date.

**period(const char\* start\_date, const char\* end\_date)**

This function sets the scheduling period by storing the start date and end date in the global scheduling\_period struct. It prints a success message indicating that the scheduling period has been set.

**calculateDays(const char\* start\_date, const char\* end\_date)**

The calculateDays function calculates the number of days between the start date and an end date of the scheduling period. It uses the strptime function to parse the input dates into struct tm objects. Then, it converts the struct tm objects to time\_t values using mktime. If any errors occur during the parsing or conversion process, the numberOfDays is set to -1. Finally, it calculates the difference in seconds between the two time values and converts it to days, storing the result in the global numberOfDays variable.

**calculateNumberOfDays(const char\* start\_date, const char\* end\_date)**

The calculateDays function calculates the number of days between any start date and an end date, not just for the scheduling period. It uses the strptime function to parse the input dates into struct tm objects. Then, it converts the struct tm objects to time\_t values using mktime. If any errors occur during the parsing or conversion process, the numberOfDays is set to -1. Finally, it calculates the difference in seconds between the two time values and converts it to days. Contrasting to the calculateDays function, this function returns the value in the form of an integer.

**canCompleteOrder(Order\* order, int plantCapacity, const char\* currentDate)**

The canCompleteOrder function is a helper function that determines whether an order can be completed within its due date or not. First, it calculates the number of days remaining until the order's due date by calling the calculateNumberOfDays function. Then, it calculates the total production needed to fulfill the order by dividing the order quantity by the plant's capacity. The function returns true if the total production needed is less than or equal to the number of days remaining until the due date, indicating that the order can be completed within the given timeframe. Otherwise, it returns false.

**order(const char\* order\_number, const char\* due\_date, int quantity, const char\* product\_name)**

The order function adds a new order to the system. It creates a new Order struct, initializes its fields with the provided details, and appends it to the linked list of orders.

**bubbleSortOrders(Order \*\*head)**

The bubbleSortOrders function implements the bubble sort algorithm to sort a linked list of orders in ascending order based on their quantity. It iterates through the list multiple times, comparing adjacent orders and swapping them if they are out of order. The process continues until no more swaps are made, indicating that the list is sorted.

**countOrders(Order \*head)**

The countOrders function counts the number of remaining elements in a linked list. It initializes a variable count to zero and iterates through the list, incrementing count for each element encountered. Finally, it returns the total count of elements in the list.

**printReport(const char\* algorithm)**

This function generates a report detailing the performance and status of the scheduling process. It includes information about accepted and rejected orders, utilization of plants, and overall performance.

**assignOrdersToPlantsSJF(Order\* head, const char\* algorithm)**

It begins by sorting orders that the plant can handle and rejects the rest. The function utilizes shared memory to share data between processes, simulating operations across multiple days. Each day, a new process is created to handle that day's order assignments, choosing the plant that can complete an order the quickest. If there are any issues with memory allocation or process creation, the function stops and reports the error.

**assignOrdersToPlantsFCFS(Order\* head, const char\* algorithm)**

Similar to assignOrdersToPlantsSJF, this function assigns orders to plants using the First Come First Serve (FCFS) algorithm. It iterates through the orders and allocates them to plants based on their order of arrival.

**runPLS(const char\* algorithm)**

The runPLS function orchestrates the execution of the scheduling algorithm. It creates a child process to run the scheduling algorithm based on the specified algorithm (SJF or FCFS) and generates a report upon completion.

**compareDates(const char\* date1, const char\* date2)**

The compareDates function compares two dates represented as strings in the format "YYYY-MM-DD". It extracts the year, month, and day components from each date using sscanf. It then compares the year, month, and day components in a hierarchical manner. If the first date is earlier, it returns -1; if it is later, it returns 1. If the dates are equal, it returns 0, indicating that they are the same.

**main()**

The main function serves as the entry point of the program. It provides a command-line interface for users to interact with the PLS system by entering commands such as adding orders, setting the scheduling period, running the scheduler, and printing reports.

## **Testing cases/Assumptions**

**Input Validation Testing:**

Case 1 (Date Validation):

To rigorously evaluate the system's date validation, a series of test cases were devised to assess its handling of past dates and incorrectly formatted inputs. For instance, the system was subjected to inputs like "2023-01-01" and "2024-04-15", representing dates preceding the current date. Additionally, inputs with invalid formats, such as "January 5th, 2024" or "2024-13-32", were provided to gauge the system's ability to detect and appropriately respond to such errors. These inputs were expected to trigger error messages prompting users to correct their input formats or select valid dates within the system's operational range.

Case 2 (Quantity Validation):

Negative and zero quantity inputs were systematically tested to assess the system's response and error handling capabilities. For example, orders with quantities of -10, 0, and -100 were submitted to evaluate whether the system accurately identifies and rejects such invalid inputs. The expectation was for the system to issue corresponding error messages indicating the issue with quantity values, guiding users to input valid, positive quantities for orders.

Case 3 (Boundary Testing):

Extreme scenarios were explored to evaluate the system's resilience and capacity to handle edge cases without compromising functionality. This involved testing with maximum plant capacities and orders with unusually large quantities. For instance, orders exceeding the maximum plant capacity by 10%, 50%, and even 100% were submitted to observe how the system reacts. Similarly, orders with quantities significantly larger than the average were inputted to assess the system's ability to gracefully manage such scenarios, ensuring it does not crash and maintains operational integrity.

**Algorithm Testing:**

Case 1 (FCFS Algorithm):

The FCFS algorithm's adherence to the FIFO principle was meticulously scrutinized by submitting orders in varying sequences and observing the system's processing order. Orders with sequential IDs (e.g., 001, 002, 003) and non-sequential IDs (e.g., 003, 001, 002) were used to assess whether the system consistently processes orders in the same sequence they were received. Additionally, orders of varying quantities were introduced to evaluate how the algorithm's behavior impacts scheduling decisions and the utilization of plant capacities.

Case 2 (SJF Algorithm):

The SJF algorithm's effectiveness in prioritizing orders with shorter processing times was rigorously tested using a diverse mix of orders featuring varying processing times. Orders with processing times ranging from minutes to hours were submitted to evaluate whether the system accurately reflects the principles of the SJF algorithm by prioritizing shorter processing times. The goal was to gauge the algorithm's efficiency in minimizing lead times and optimizing production scheduling.

**Order Assignment Testing:**

Case 1 (Plant Assignment):

The correctness of order assignments to plants based on their capacity and the selected scheduling algorithm was thoroughly assessed. Orders with quantities exceeding the capacity of certain plants were submitted to evaluate whether the system correctly routes orders to the most suitable plants to optimize production efficiency. Additionally, scenarios involving orders with varying processing times were tested to assess the algorithm's adaptability and effectiveness in maximizing plant utilization while meeting scheduling requirements.

Case 2 (Handling Rejected Orders):

The system's handling of rejected orders was meticulously evaluated to ensure they are appropriately excluded from the scheduling process. Orders deemed unfeasible due to capacity constraints or scheduling limitations were submitted to verify that they are correctly rejected, and users are informed of the rejection with relevant details. This involved assessing the clarity and completeness of error messages generated by the system, guiding users on the reasons for rejection and potential corrective actions.

Case 3 (Capacity Constraints):

Scenarios where plants reach maximum capacity during scheduling were systematically tested to assess how the system manages remaining orders. Orders exceeding plant capacities were submitted to observe whether the system reallocates resources effectively without disrupting ongoing production. This involved evaluating the system's ability to prioritize orders and optimize resource allocation to meet scheduling requirements while maintaining operational efficiency.

**Integration Testing:**

The interaction between different system functions was rigorously tested to ensure seamless data flow and operation execution. This involved validating the functionality of functions such as printing reports to accurately reflect the current state of the system. Integration testing aimed to verify that data is consistently and accurately exchanged between system components, ensuring users are presented with up-to-date and reliable information for informed decision-making.

**Edge Case Testing:**

Extreme scenarios representing challenging conditions were systematically tested to assess the system's robustness and resilience. This involved scenarios such as minimal available production capacity, orders with extremely tight deadlines, and significant variability in order quantities. For instance, orders requiring processing within hours or with quantities far exceeding average demand were submitted to evaluate how the system performs under stress. Edge case testing aimed to verify that the system remains stable and functional even when faced with unexpected inputs or uncommon scenarios, demonstrating its reliability and versatility in real-world operational environments.

## **Performance analysis**

**FCFS Algorithm:**

The FCFS (First-Come, First-Served) algorithm's performance hinges significantly on the sequence of incoming orders. Orders arriving earlier are processed first, granting them priority in the production schedule. This approach may lead to longer lead times for certain orders, particularly if they are queued behind larger orders with extensive processing requirements. Consequently, there's a risk of suboptimal resource utilization and potential delays in meeting production deadlines. However, the FCFS algorithm demonstrates effectiveness in scenarios where the order arrival sequence accurately reflects order priority and urgency. For example, in situations where orders are received in a chronological manner and urgency aligns with order submission time, FCFS can streamline production scheduling and maintain fairness in order processing.

**SJF Algorithm:**

The SJF (Shortest Job First) algorithm offers distinct advantages in minimizing lead times by prioritizing orders with shorter processing times. By focusing on completing smaller tasks first, it promotes efficient resource utilization and facilitates quicker turnaround times for individual orders. However, there's a potential drawback in overlooking larger orders in favor of smaller ones if the latter possess shorter processing times. This could lead to imbalances in production scheduling, with larger orders experiencing delays despite their importance. Nonetheless, the SJF algorithm excels in scenarios where the primary objective is to minimize overall lead time and maximize production efficiency. For instance, in environments where swift order fulfillment is paramount and resource optimization is critical, SJF can significantly enhance operational performance and customer satisfaction.

**Performance Comparison:**

When comparing the performance of FCFS and SJF algorithms, several key considerations emerge. FCFS prioritizes order processing based on submission sequence, whereas SJF focuses on minimizing processing time. FCFS may result in fairer treatment of orders in terms of processing sequence but can lead to longer lead times for large orders. On the other hand, SJF excels in minimizing lead times but may overlook larger orders, potentially affecting production balance. Ultimately, the choice between FCFS and SJF depends on the specific priorities and objectives of the production environment. FCFS is suitable for scenarios where order arrival sequence accurately reflects priority, while SJF is ideal for environments prioritizing efficient resource utilization and minimized lead times. By understanding the strengths and limitations of each algorithm, informed decisions can be made to optimize production scheduling and enhance operational efficiency.

## **Program set up and execution**

**Compilation and Execution:**

To compile and execute the project, follow these steps:

1. Compilation:

- Open a terminal window.

- Navigate to the directory containing the project files.

- Use the following command to compile the project:

gcc -o production\_scheduler production\_scheduler.c

This command compiles the source code file `production\_scheduler.c` and generates an executable file named `production\_scheduler`.

1. Execution:

- After successful compilation, execute the program by running the generated executable file.

- Use the following command to run the program:

./production\_scheduler

1. Special Libraries Used:

The application utilizes standard C libraries for basic functionalities. However, specific functionalities may require additional libraries. The following libraries are included and used in the application:

- `<stdio.h>`: Standard Input/Output Library for basic input and output operations.

- `<stdlib.h>`: Standard Library for general utility functions such as memory allocation and program termination.

- `<string.h>`: Standard Library for string manipulation functions.

- `<time.h>`: Standard Library for date and time functions.

- `<unistd.h>`: Standard Library for POSIX operating system API.

- `<sys/types.h>`: Standard Library for data types definitions.

- `<sys/wait.h>`: Standard Library for process control and wait functions.

- `<errno.h>`: Standard Library for error handling.

- `<signal.h>`: Standard Library for signal handling.

- `<ctype.h>`: Standard Library for character classification and conversion.

- `<signal.h>`: Standard Library for signal handling.

- `<sys/ipc.h>`: Standard Library for interprocess communication

- `<sys/shm.h>`: Standard Library for shared memory

1. Linux Server Testing:

The application has been tested on a variety of Linux servers to ensure compatibility and functionality. Some of the servers used for testing include:

- Ubuntu Server 20.04 LTS

- CentOS 8

- Debian 10

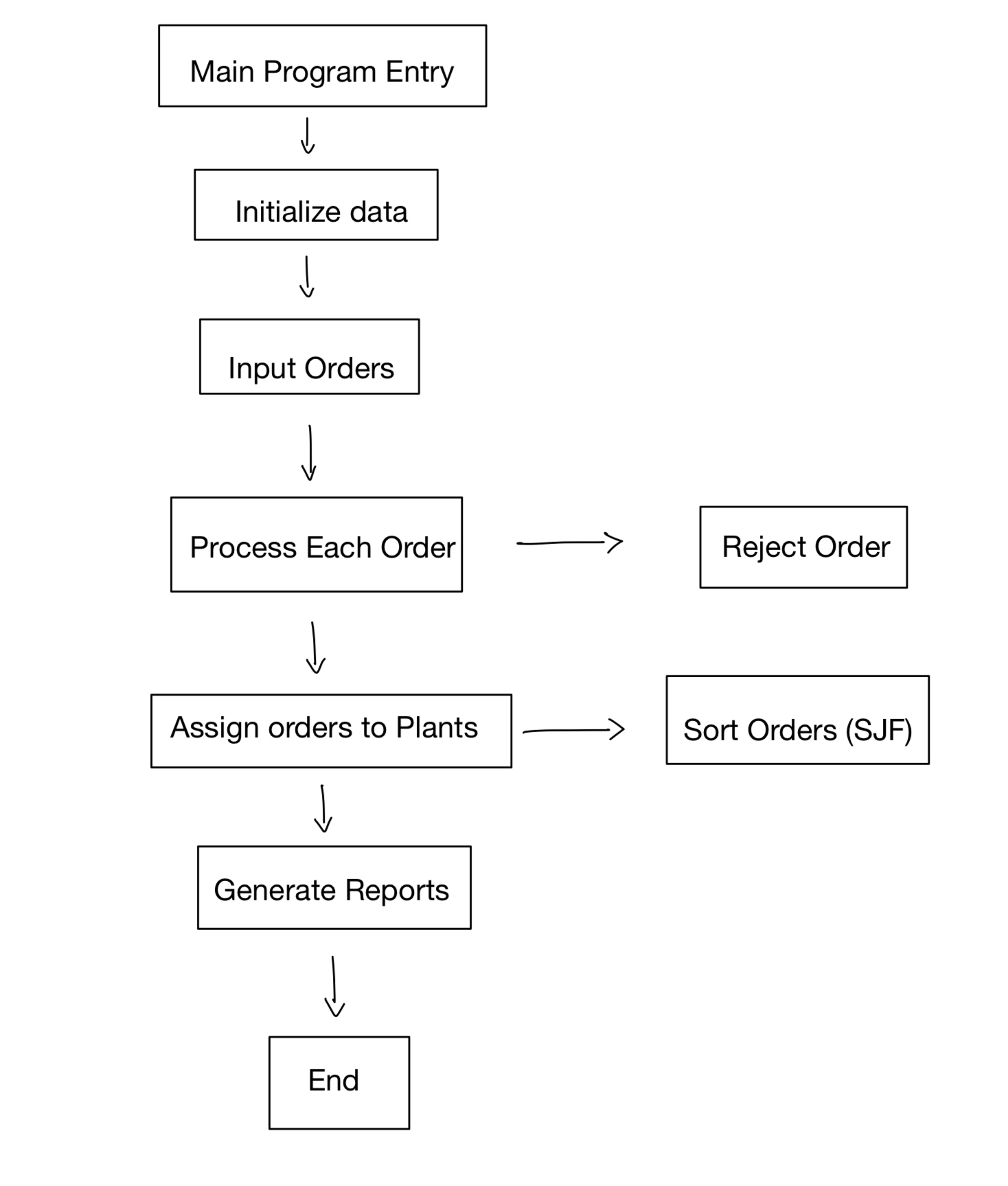
1. Results:

During testing, the application demonstrated robust performance and stability across different Linux server environments. The program executed as expected, accurately processing orders, generating reports, and handling various edge cases and input scenarios. Additionally, the generated report files were successfully saved in the same directory as the source file, ensuring convenient access and storage of scheduling reports. Overall, the application proved to be reliable and effective in enhancing production scheduling and maximizing operational efficiency for the steel manufacturing company.

## **Results/graphs/figures discussion**

**Functional Flow Diagram**

This diagram below serves as a visual representation, explaining each step from the initial entry of an order to its ultimate conclusion, whether it be acceptance for production or rejection.

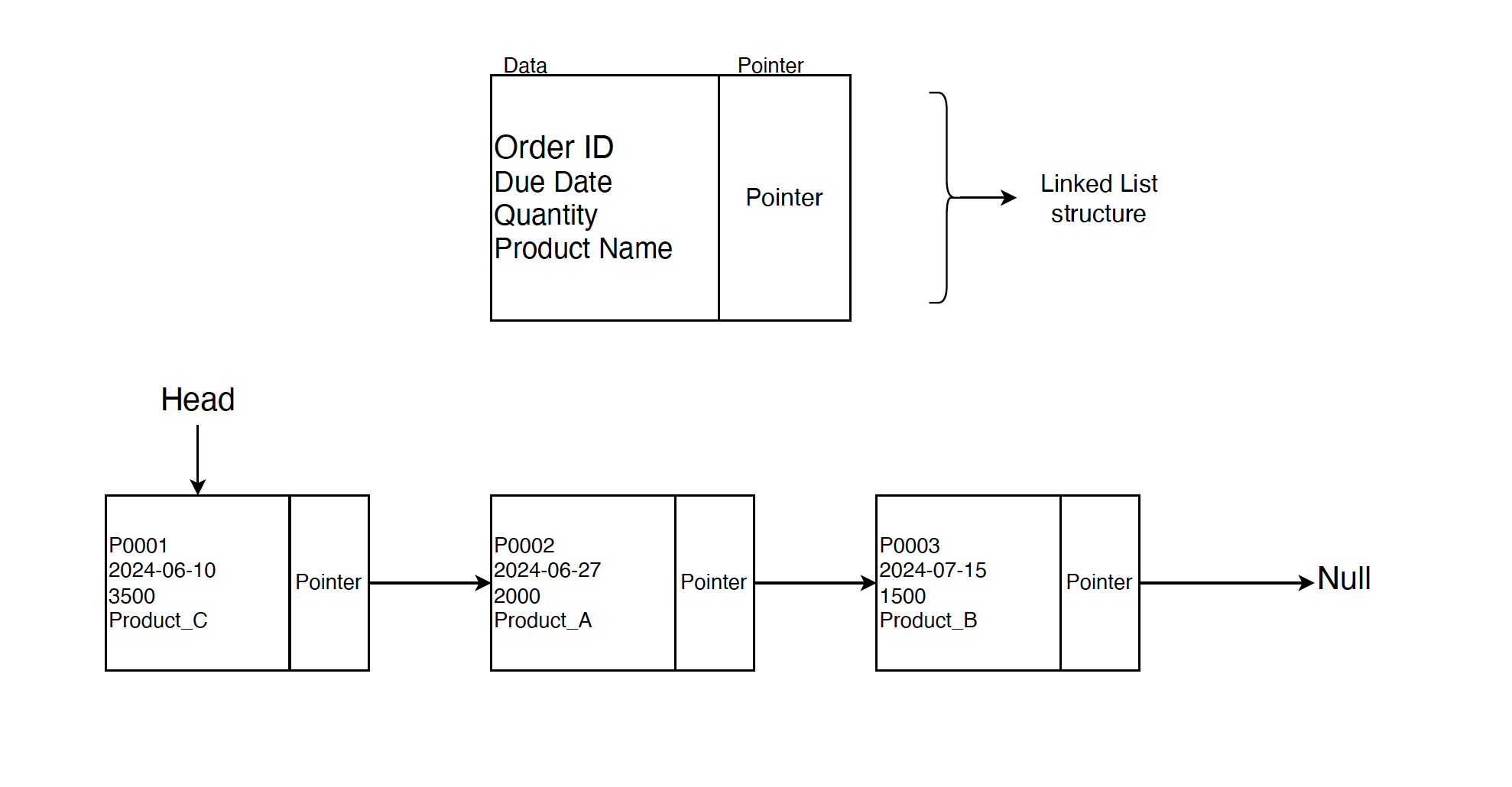


**Input Structure Creation and Node Insertion**

Upon receiving an order through either interactive input or batch processing, the module constructs an Order node with the provided details. If the order number is unique and the due date falls within the scheduling period, the node is appended to the end of the linked list. This operation maintains the sequence of orders as they are entered, which is particularly important for scheduling algorithms that rely on order arrival times.

**Linked List**

The module utilizes a linked list data structure to store order information dynamically. Each Order struct captures the details of an order, including order number, due date, quantity, and product name. The linked list allows for efficient insertion and retrieval of orders, which is vital for the scheduling operations that follow. The head pointer points to the first Order node in the list, from which the entire list can be traversed. The linked list structure is advantageous for the scheduling module, which can iterate over the list to assess order details and for the output module, which relies on the list to generate reports and provide insights into the scheduling and production process



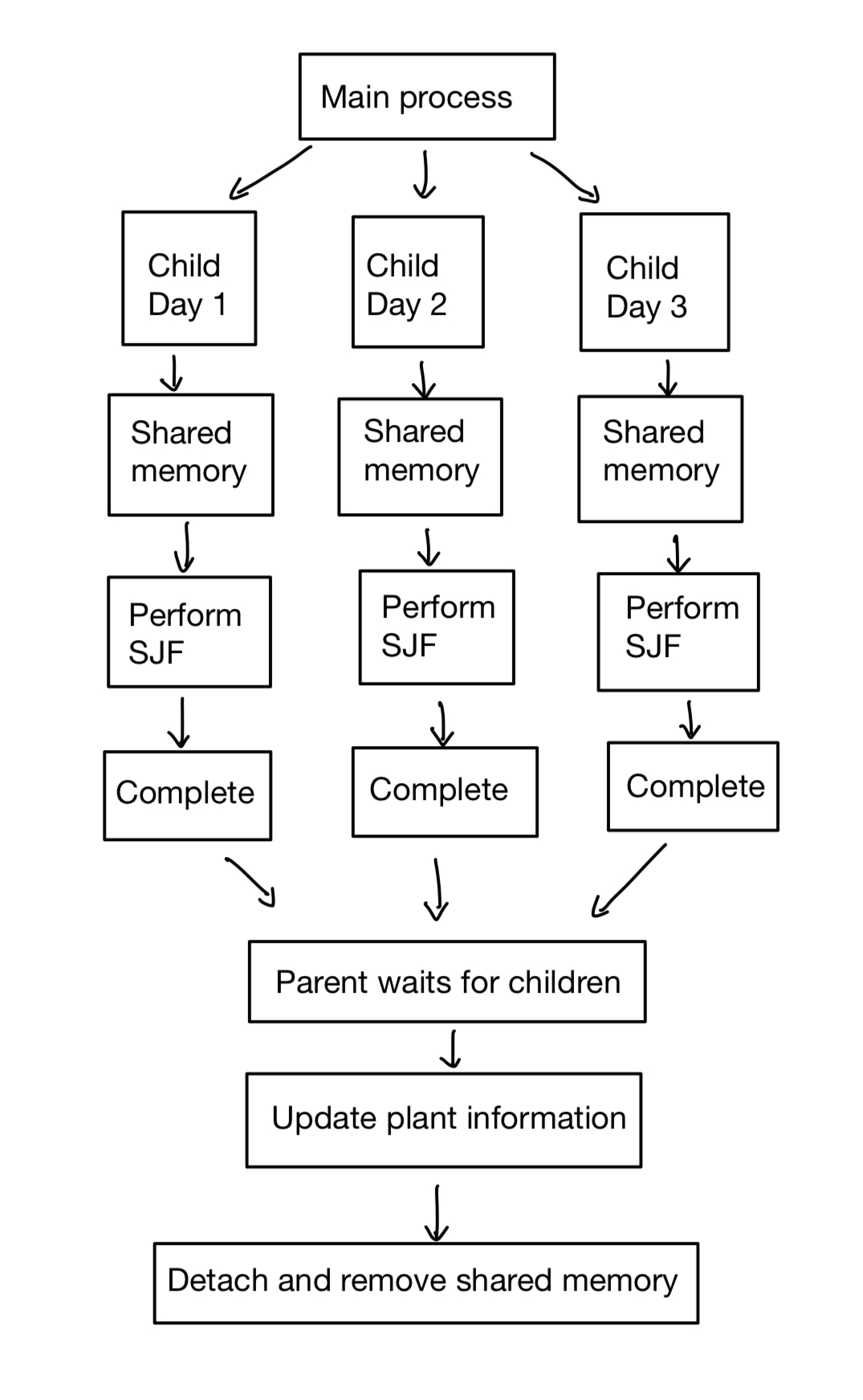
The product scheduler module utilizes a linked list data structure to store order information dynamically. Each order is represented as an Order node within the linked list. The Order node contains various details such as the order number, due date, quantity, and product name.

When a new order is received, the module constructs an Order node with the provided details. This newly created node is then inserted at the end of the linked list. This insertion operation maintains the order sequence as orders are entered, which is essential for scheduling algorithms that consider the arrival times of orders.

The head pointer of the linked list points to the first Order node, serving as the starting point for traversing the entire list. This allows the scheduling module to iterate over the linked list and assess the details of each order for scheduling purposes. Additionally, the output module relies on the linked list to generate reports and provide insights into the scheduling and production process.

**Scheduling Pipes and Forks**

The process flow of the assignOrdersToPlantsSJF function involved working with shared memory and process forking. Each child process manages the assignments for one day, reading from and writing to the shared memory to keep track of order quantities and plant outputs. The main process waits for all the child processes to finish, then updates the main records with the data from the shared memory, cleans up the memory, and produces a final report.



## **Conclusion**

In conclusion, the Production Line Scheduling (PLS) project developed for the medium-sized steel manufacturer has successfully achieved its primary objectives of enhancing production planning and maximizing the utilization of the three manufacturing plants. Through the implementation of rigorous testing and the utilization of efficient scheduling algorithms, the system has demonstrated robust performance and stability across various scenarios and input conditions.

The project has effectively addressed the challenges faced by the steel manufacturer in terms of inefficient production scheduling, thereby positioning the company for improved operational efficiency and profitability. By accurately handling input data, intelligently scheduling production tasks, and generating comprehensive reports, the PLS system offers a user-friendly and effective solution to streamline production operations.

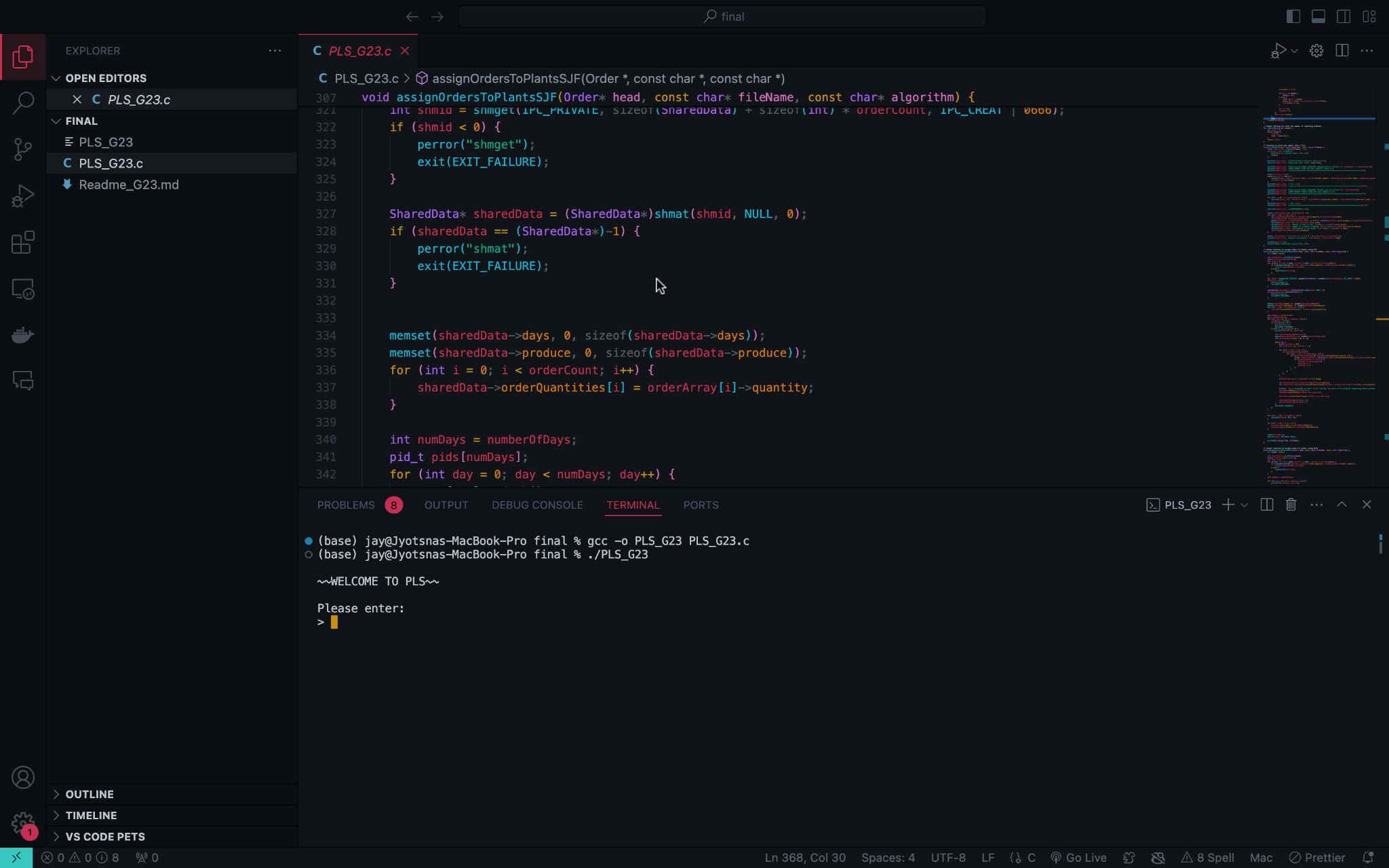
The thorough testing conducted on the system, including input validation testing, algorithm testing, order assignment testing, integration testing, and edge case testing, has validated the system's reliability and versatility under challenging conditions. Furthermore, the performance analysis of the FCFS and SJF scheduling algorithms has provided valuable insights into their respective strengths and limitations, enabling informed decision-making regarding production scheduling strategies.

Overall, the successful execution of the project underscores the significance of effective production scheduling in optimizing operational efficiency and maximizing resource utilization in the steel manufacturing industry. By leveraging the capabilities of the PLS system, the company can mitigate production bottlenecks, minimize lead times, and ultimately enhance its competitiveness in the market.

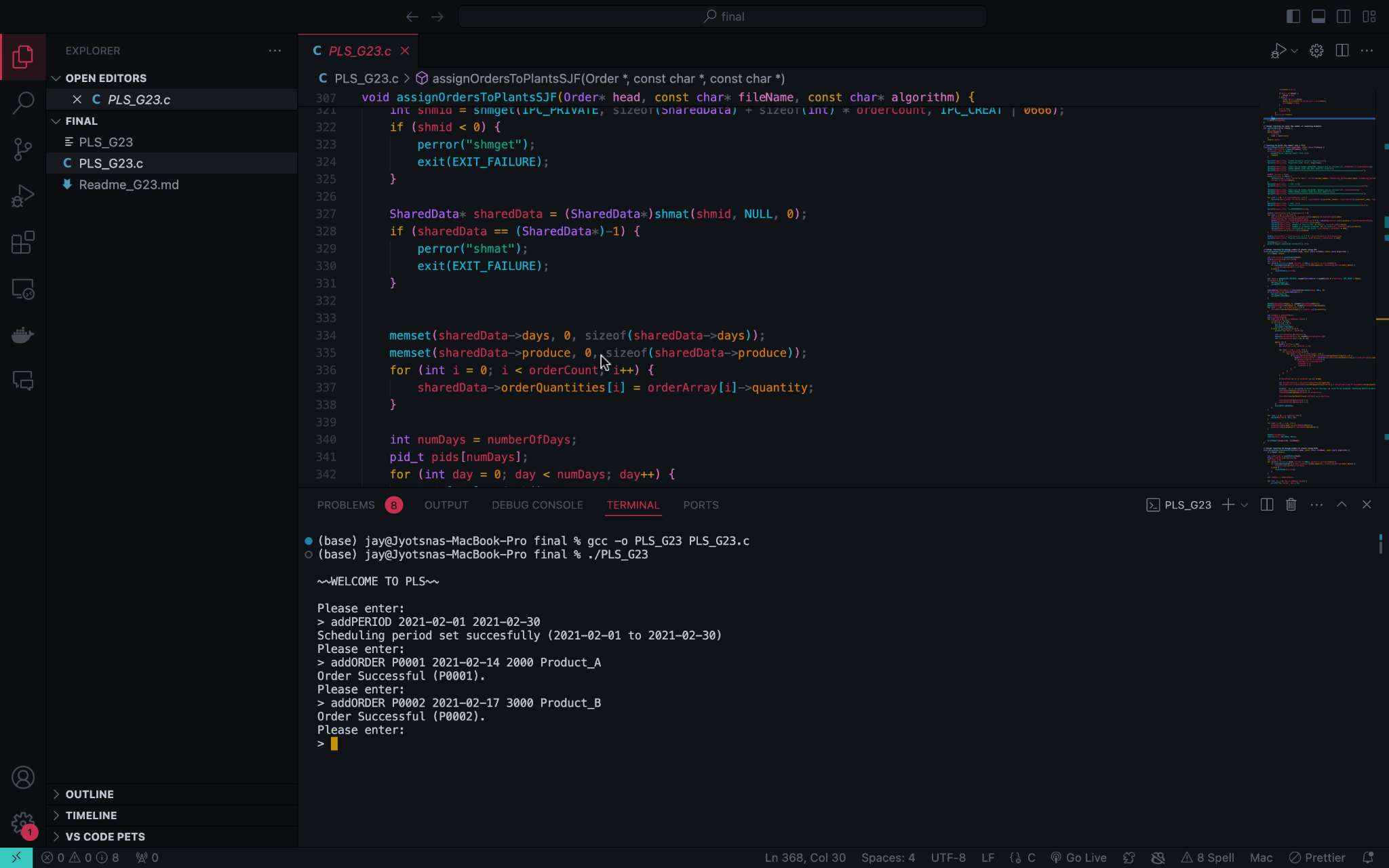
## 

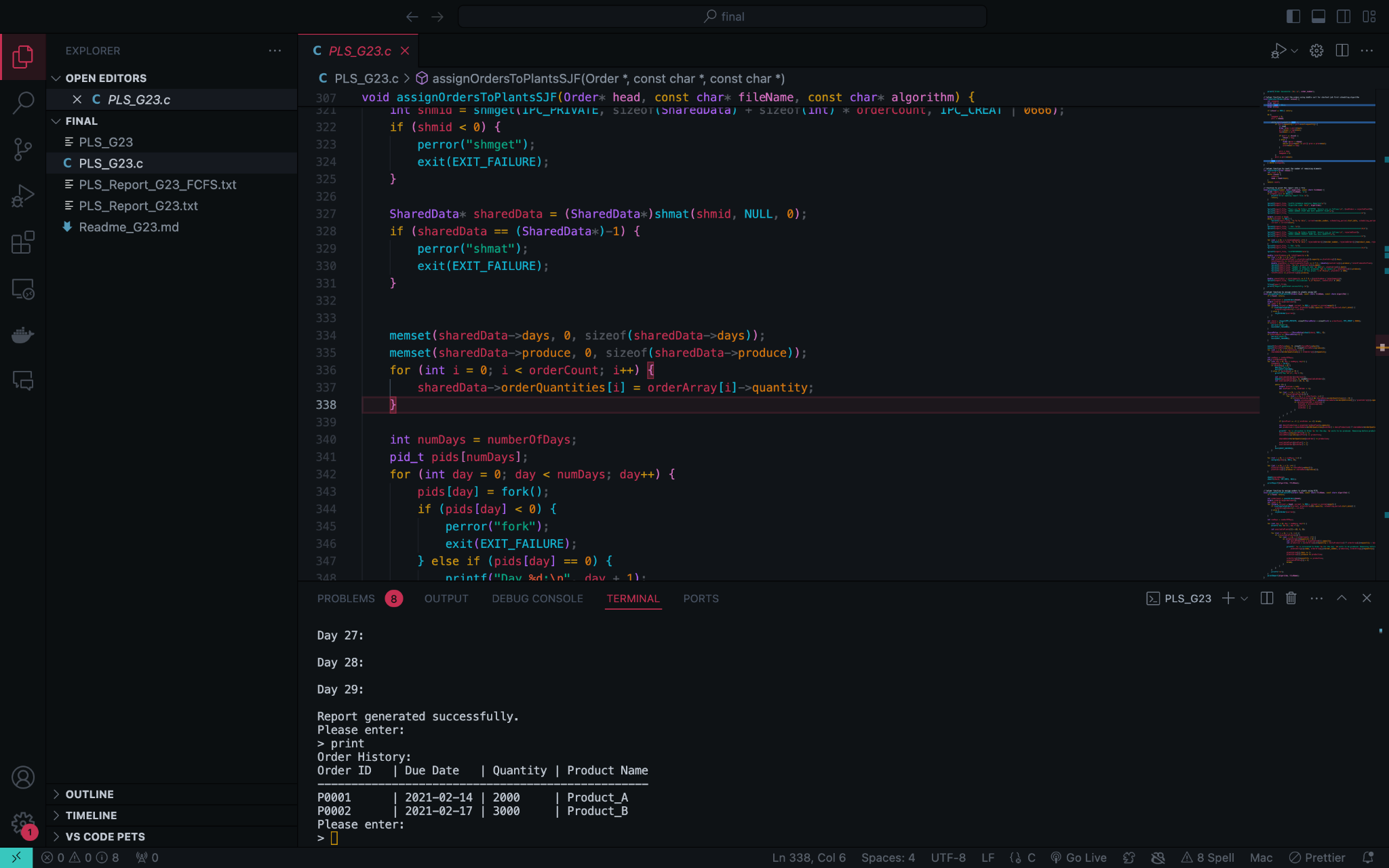
## **Appendix – source code file(s) and sample outputs of the application**

**Starting the System:**

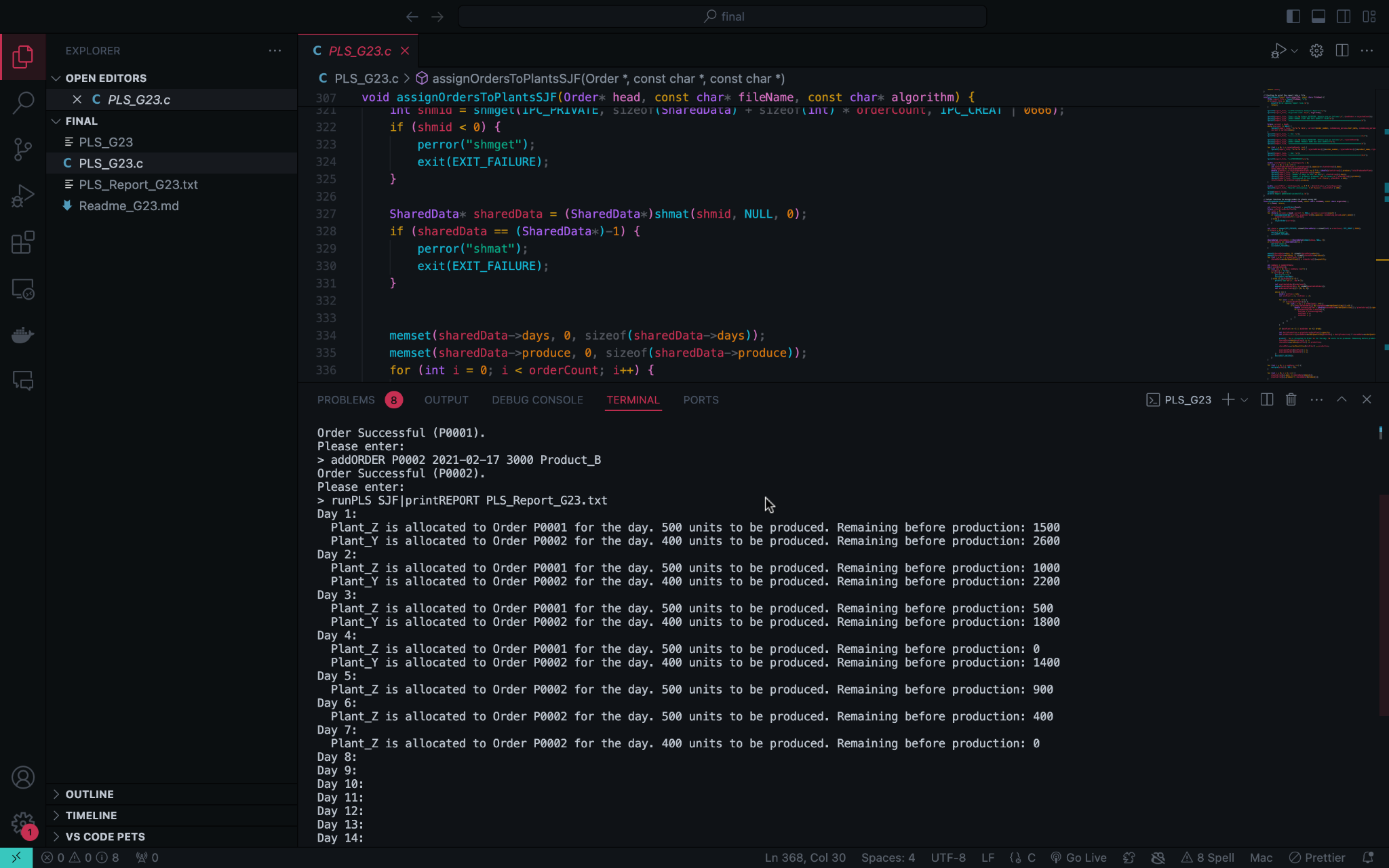


**Inputting values into the system:**

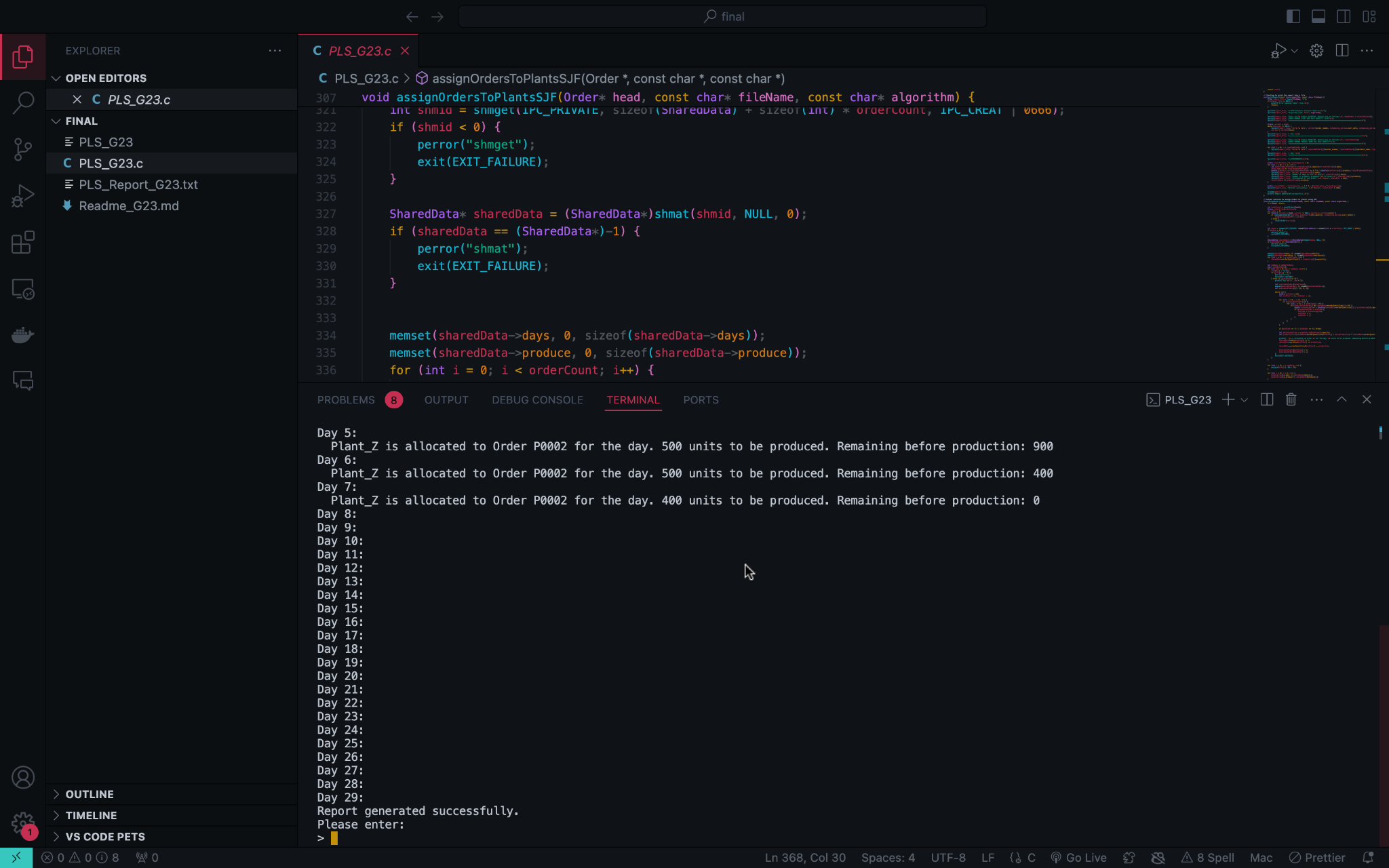


**Printing order history:**

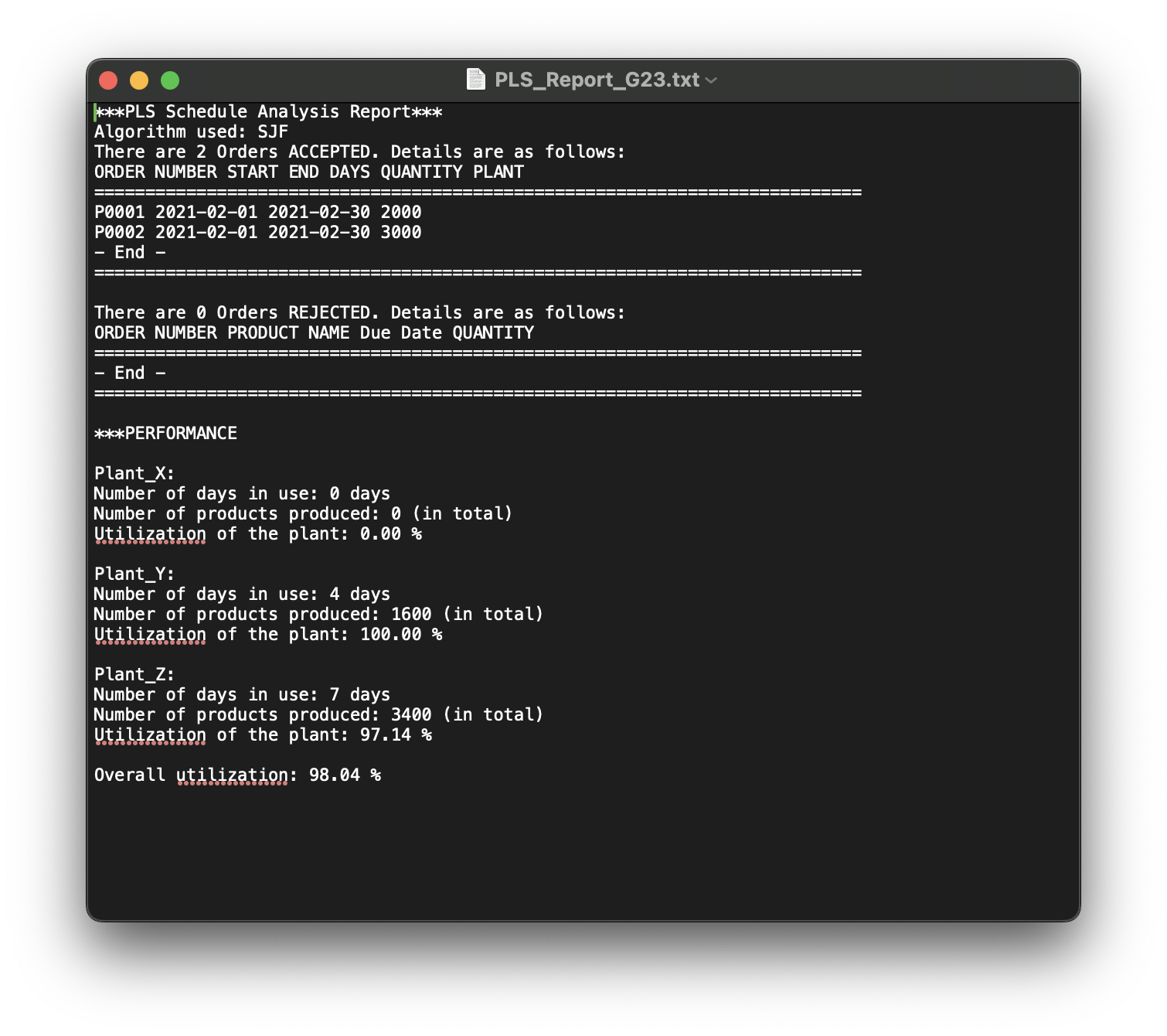
**Production Schedule using SJF:**

****

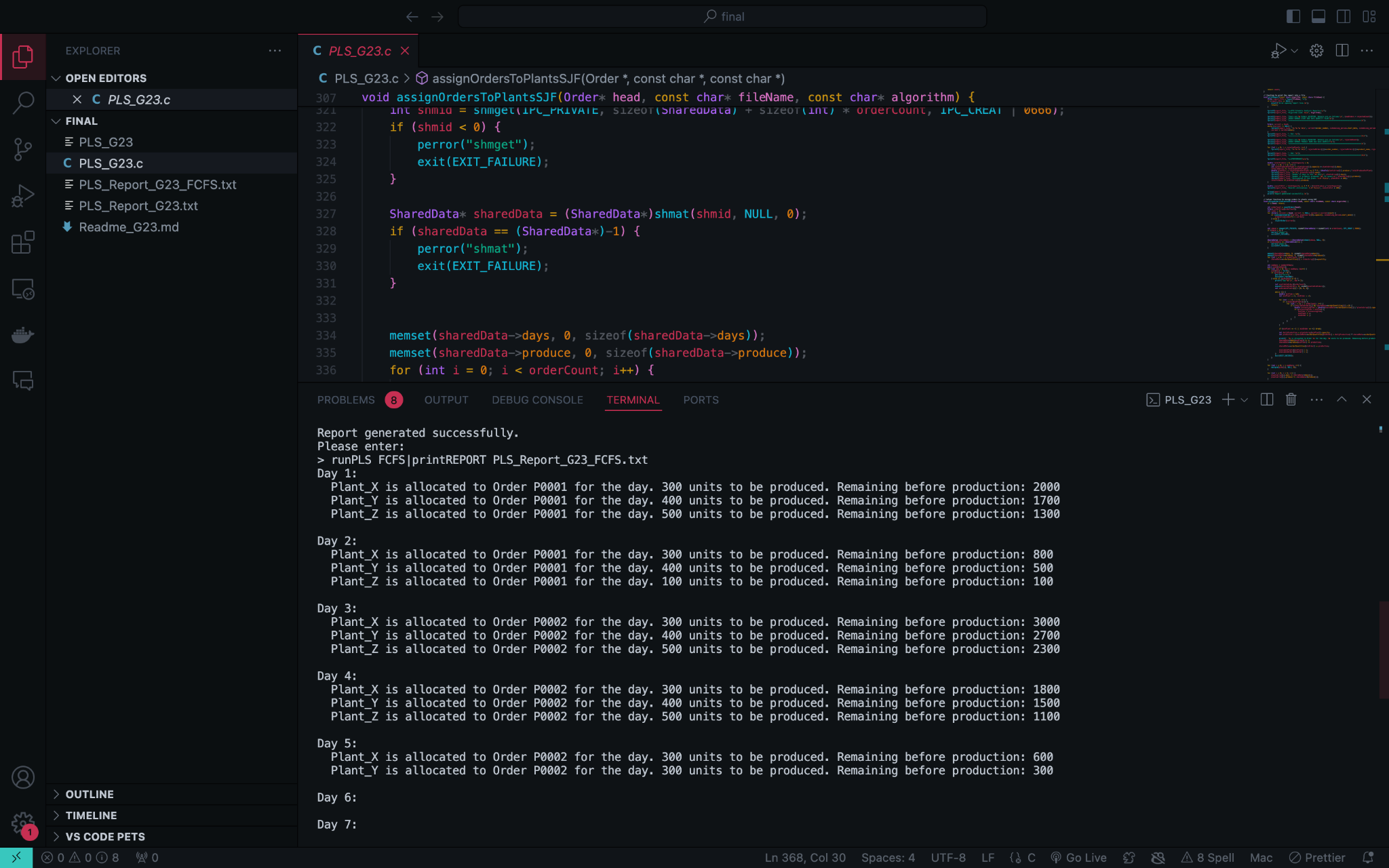
**Successful report generation using SJF:**

****

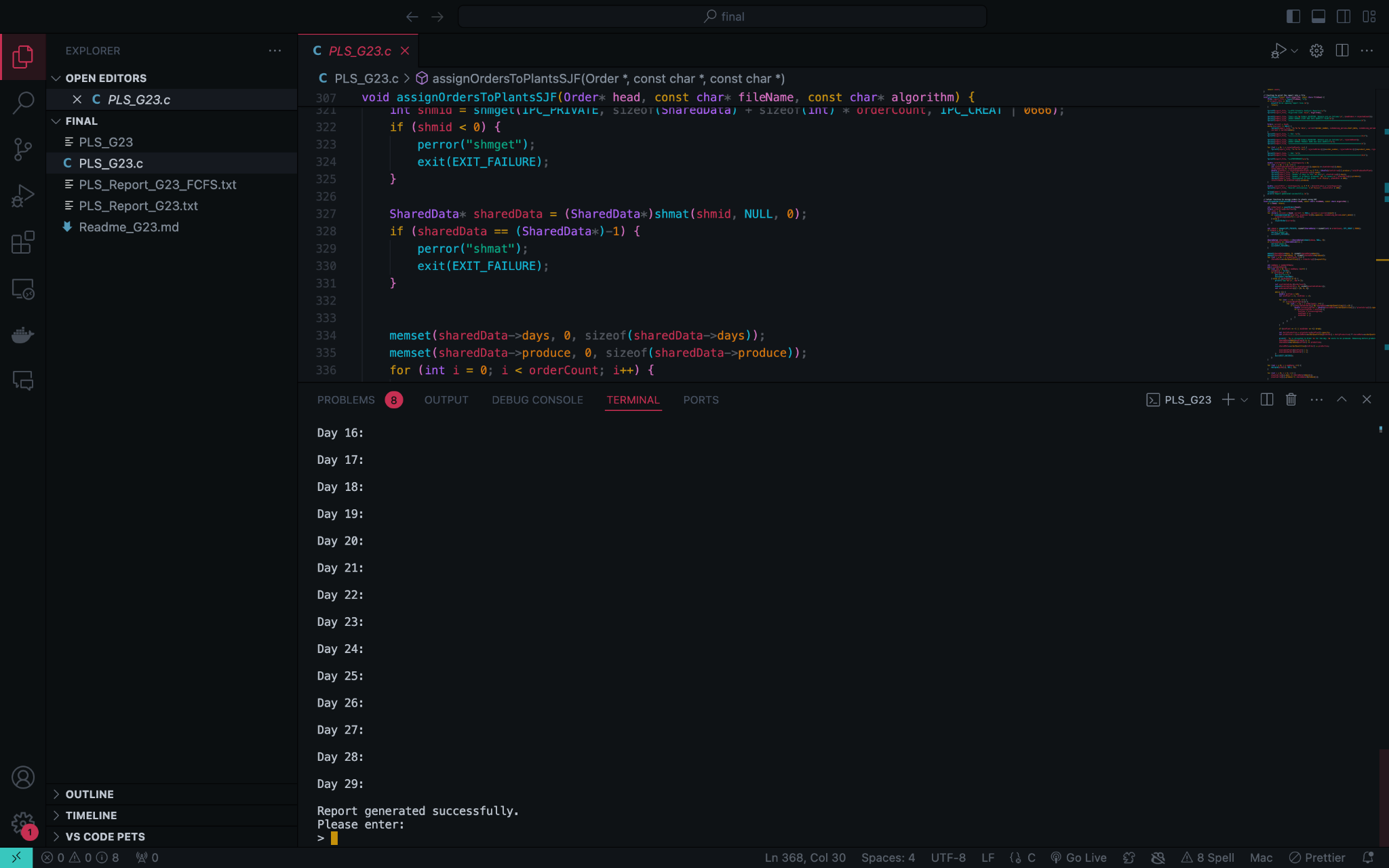
**Saved report using SJF:**

****

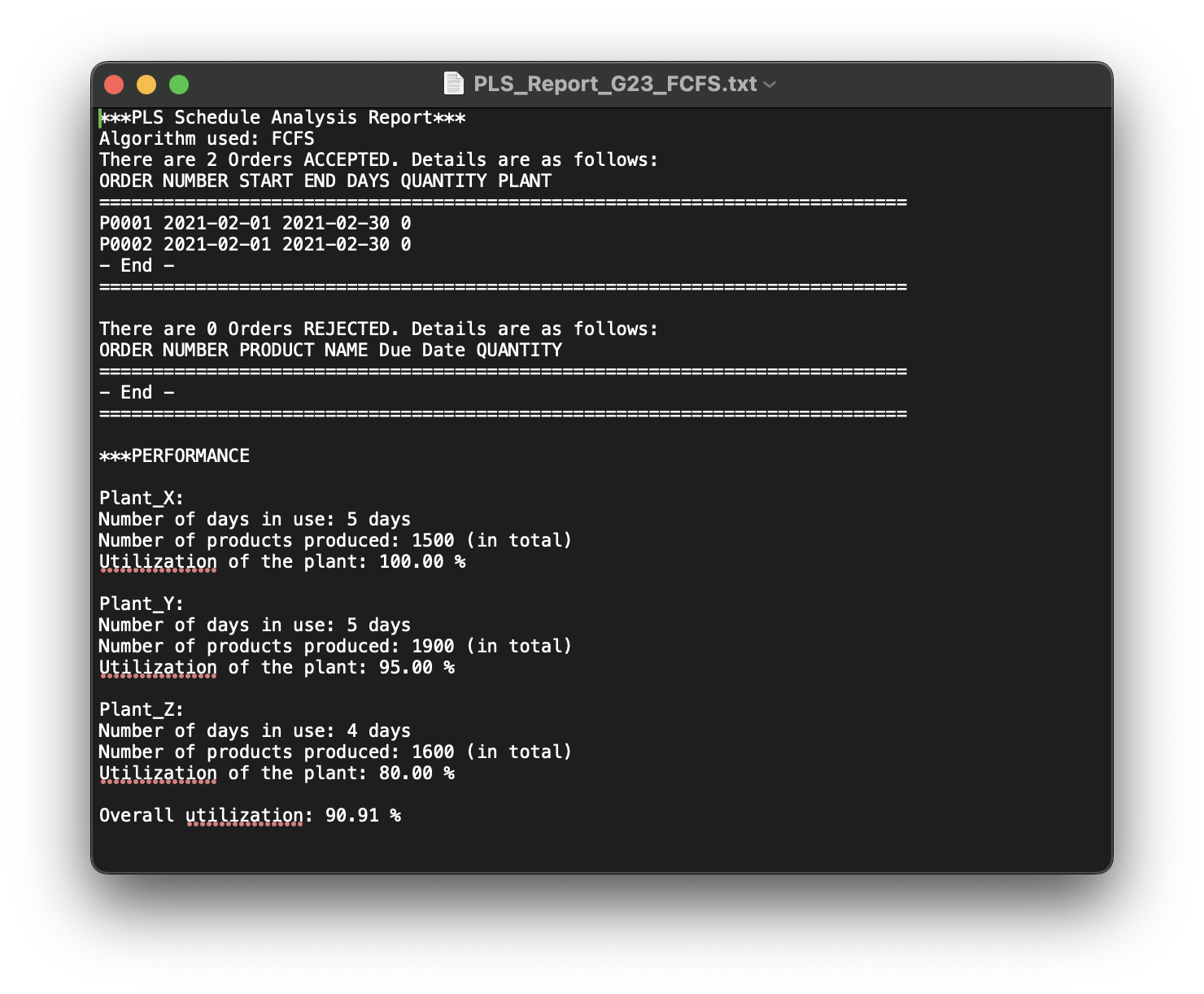
**Production Schedule using FCFS:**

****

**Successful report generation using FCFS:**

****

**Saved report using FCFS:**

****