COMP2432 Group Project: Steel-making Production Line Scheduler (PLS)

1st Yuhang DAI
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
22097845d@connect.polyu.hk

2nd Zhanzhi LIN
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
22097456d@connect.polyu.hk

3rd Zirui KONG
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
22103493d@connect.polyu.hk

4th Zhaoyu CUI
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
22102947d@connect.polyu.hk

5th Qinye ZHANG
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
22098835d@connect.polyu.hk

***Abstract*—This report reviews the creation and assessment of our production line scheduler (PLS) tailored for a medium-sized steel-making manufacturer, aimed at tackling inefficiencies in scheduling and utilization of multiple plants. The significance of CPU scheduling algorithms lies in their ability to effectively manage CPU resources, ensuring maximum utilization and optimal system performance by determining the sequence and timing of process execution. To simulate the process of CPU scheduling, our application, developed in C and executed in a Linux environment, dynamically accommodates new orders and scheduling parameters, evaluates real-time plant capacities, and generates detailed analytics on order statuses and productivity levels. By applying different CPU scheduling algorithms to real-world scenarios, we hope to visualize their performance and intuitively compare the advantages and disadvantages of each algorithm.**

***Index Terms*—CPU Scheduling Algorithms, Operating Systems**

I. INTRODUCTION

Modern computing systems feature multiple components such as processors, input/output devices, and main memory, forming a complex architecture that requires sophisticated management. The operating system (OS) plays a crucial role as the regulator of these resources, coordinating their control and allocation to optimize system performance by switching between kernel mode and user mode during program execution [3]. Processes, which are essentially programs in execution, rely on the OS to allocate CPU time to complete their tasks. The advent of multiprogramming and multitasking in operating systems is a significant reason why modern computer systems require efficient CPU scheduling, as these systems execute multiple program tasks concurrently. According to Silberschatz et al. in ”Operating System Concepts” [7], multiprogramming’s primary objective is maximizing resource utilization by assigning the CPU to other processes when the current process is idle.

To facilitate this, the scheduler, a crucial piece of system software, manages the allocation of resources by organizing queued requests. It is categorized into three distinct types: the long-term scheduler, which admits processes from the storage on the hard disk (HDD) into the Random Access Memory (RAM), effectively deciding which processes enter the system; the short-term scheduler, which selects processes from the ready queue to be executed next on the CPU; and the medium-term scheduler, which temporarily removes processes from active contention for the CPU to manage the level of multiprogramming and reduce CPU overhead [7].

Our project’s primary aim is to simulate the complexities of scheduling algorithms in a real-world manufacturing context, specifically tailored for a medium-sized steel-making manufacturer experiencing inefficiencies in scheduling and plant utilization. The motivation behind this work is to systematically explore and apply CPU scheduling algorithms, traditionally used in computing, to optimize the production schedules of three plants with varying output capacities. In our project, the Input Module functions similarly to the job queue in an operating system, where it collects and organizes incoming tasks (in this case, production orders) before they are processed. Like the job queue, this module ensures all necessary information, such as order due date, quantity, and product name, is available to properly schedule tasks according to system capabilities and constraints. The Scheduling Kernel then takes these inputs to generate optimal production schedules, paralleling the role of the scheduler in an OS that assigns CPU time to various processes based on selected algorithms. It employs multiple algorithms to handle the scheduling tasks effectively:

- **First-Come, First-Served (FCFS):** This algorithm schedules tasks based on the order number of requests, assigning factories sequentially to the orders as they come in, mimicking the FCFS approach in operating systems where tasks are processed in the order of their arrival.
- **Shortest Job First (SJF):** Here, the algorithm prioritizes orders based on the quantity of the product, with

smaller quantities being scheduled first, which is similar to the SJF approach in operating systems that prioritize processes with shorter CPU burst times.

- **Our Novel Scheduling Algorithm:** A new algorithm developed for this project, which prioritizes orders based on larger quantities, aiming to execute high-volume orders sooner to optimize throughput and resource allocation.

The Output Module displays the allocation of these tasks, akin to system logs that provide a clear breakdown of resource allocation over time for operational monitoring. The Analyzer Module outputs detailed metrics on plant utilization, akin to system monitoring tools in operating systems that assess and report on resource usage. Specifically, it calculates the number of days each plant (X, Y, and Z) is in use and the total products produced during these days. Utilization percentages are then derived from these figures, providing insights similar to those offered by performance counters in an OS that track and analyze CPU usage, disk reads/writes, and other system resources.

The project report is structured to provide a clear and comprehensive analysis of the scheduling system simulation. It will begin with relevant operating systems concepts, such as CPU scheduling algorithms, which inform the methodologies used in the project. The novel scheduling algorithms employed will alao be introduced. The software structure is described in Section VI, providing insights into the architectural choices. This will be followed by several testing cases and assumptions, in order to foster the understanding of how algorithms are implemented in this project. In Section VIII, a thorough performance analysis of each implemented scheduling algorithm is presented. Section IX functions as a user manual, explaining the compilation and execution procedures of the project, as well as the specifics of necessary libraries and the Linux server environment used. Then, the report will present results of different cases alongside graphs and figures. At last, it will conclude in Section XI, synthesizing all insights and expressing our perspectives.

II. Related Work

CPU Scheduling has always been a vital task in multiprogramming systems, and a considerable amount of attempts have been made to increase the efficiency of scheduling algorithms. Goel and Garg [4] provide a detailed examination of CPU scheduling algorithms' design, effectiveness, and suitability for different types of systems and situations, and discuss the characteristics of each scheduling algorithm, including FCFS, Shortest Job First, Round Robin, and Priority Scheduling, using comparative analysis to highlight their respective advantages and limitations. Another study [5] presents an analysis of various simple and heuristic scheduling algorithms using a theoretical model of a multiprogramming system. The paper introduces a new heuristic scheduling algorithm that utilizes a look-ahead strategy, showing its superior performance over simpler algorithms through worst-case performance comparisons. Additionally, different algorithms are compared on the basis of six parameters: waiting time, response time, throughput, fairness, CPU utilization, starvation, preemption, and predictability [1].

Apart from comparing the pros and cons of existing algorithms, researchers also proposed an optimized round-robin scheduling algorithm aimed at improving CPU efficiency in real-time and time-sharing operating systems, illustrating the limitations of traditional round-robin scheduling, such as high context switch rates and long waiting times, and introducing a modified approach that reduces these inefficiencies, enhancing overall system throughput [8]. Rajput and Gupta [6] explored a hybrid scheduling algorithm that combines the benefits of round-robin and priority scheduling, incorporating a method to adjust priorities dynamically (known as aging).

III. Concept

Numerous CPU scheduling algorithms exist, each with distinct characteristics, and choosing a specific algorithm can benefit some types of processes more than others. It is crucial to evaluate the properties of the various algorithms available to select an appropriate algorithm for a given situation.

A. First-Come, First-Served Scheduling

The First-Come, First-Served (FCFS) scheduling algorithm is the most straightforward method for CPU scheduling [4]. In this approach, the first process to request the CPU is the first to receive CPU access. This policy is efficiently implemented using a FIFO (First-In, First-Out) queue. As processes arrive, they are added to the end of the queue through their process control block (PCB). When the CPU becomes available, it is assigned to the process at the front of the queue, which is then removed upon starting execution [7]. To implement the FCFS algorithm, we need to calculate the waiting time, turn-around time. A simple program using FCFS algorithm is presented below:

FCFS Scheduling of processes with different arrival times:

```
#include<stdio.h>
#include<stdlib.h>

// Structure for processes
//with all the necessary time values
typedef struct Process {
    int id, bt, at, ct, tat, wt;
} Process;

// Function prototypes
void input(Process *, int);
void calculate(Process *, int);
void display(Process *, int);
void sort(Process *, int);

int main() {
    int n;
    printf("\nEnter the number of processes:\n");
    scanf("%d", &n);
    Process *p = (Process*) malloc(n * sizeof(
    Process));

    input(p, n);
    sort(p, n);
    calculate(p, n);
    display(p, n);
```

```c

    free(p);
    return 0;
}

void input(Process *p, int n) {
    for (int i = 0; i < n; i++) {
        printf("\nAT of P%d:\n",i+1);
        scanf("%d", &p[i].at);
        printf("\nBT of P%d:\n",i+1);
        scanf("%d", &p[i].bt);
        p[i].id = i + 1;
    }
}

void calculate(Process *p, int n) {
    int sum = 0;
    sum += p[0].at;
    for (int i = 0; i < n; i++) {
        sum += p[i].bt;
        p[i].ct = sum;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        if (i+1<n && sum<p[i + 1].at) {
            sum = p[i + 1].at;
        }
    }
}

void sort(Process *p, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j=0; j < n-i-1; j++) {
            if (p[j].at > p[j + 1].at) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

void display(Process *p, int n) {
    printf("P AT BT WT TAT CT\n");
    for (int i = 0; i < n; i++) {
        printf(" P[%d] %d %d %d %d %d\n",
        p[i].id, p[i].at, p[i].bt,
        p[i].wt, p[i].tat, p[i].ct);
    }
}
```

Listing 1. FCFS Example

This C program implements the First-Come, First-Served (FCFS) scheduling algorithm, used in operating systems to manage process execution in the order of their arrival. It starts by defining a ‘Process‘ struct to store essential information such as process ID, burst time, arrival time, completion time, turnaround time, and waiting time. The main function allocates memory for an array of ‘Process‘ structures based on the number of processes entered by the user, then invokes functions to input process data, sort them by arrival time, calculate scheduling times, and display the results. The ‘input‘ function collects arrival and burst times from the user, while the ‘sort‘ function orders processes using a bubble sort to ensure they are scheduled according to their arrival times, adhering to the FCFS principle. The ‘calculate‘ function computes each process’s completion, turnaround, and waiting times by sequentially adding each process’s burst time to a running sum, adjusting for any gaps between processes. Finally, the ‘display‘ function outputs the scheduling details in a tabular format. This program exemplifies a simple, non-preemptive scheduling algorithm without priorities or interruptions, providing a foundational understanding of process scheduling in operating systems.

B. *Shortest-Job-First Scheduling*

The Shortest-Job-First (SJF) scheduling algorithm, also known as the shortest-next-CPU-burst algorithm, is an approach used in CPU scheduling that prioritizes processes based on the duration of their forthcoming CPU burst rather than their total duration [3]. This algorithm is designed to allocate the CPU to the process with the shortest upcoming CPU burst when the CPU becomes available. If two processes have equal next CPU bursts, First-Come, First-Served (FCFS) scheduling is applied to break the tie. SJF has the distinct advantage of providing the minimum average waiting time among all scheduling algorithms and is considered a Greedy Algorithm.

However, SJF scheduling can lead to potential issues such as starvation, where longer processes might never get executed if shorter ones continue arriving [2]. This problem can be mitigated by implementing the concept of ageing, which gradually increases the priority of waiting processes [2]. Despite its efficiency, SJF is often considered impractical for general-purpose operating systems since it requires precise knowledge of future CPU bursts, which are typically unpredictable [3]. Nevertheless, execution times can sometimes be estimated using methods like the weighted average of previous execution times, making SJF viable in specialized environments where accurate estimates of running time are feasible.

```c
#include <stdio.h>
int main()
{
  int A[100][4];
  int i, j, n, total = 0, index, temp;
  float avg_wt, avg_tat;
  printf("Enter number of process: ");
  scanf("%d", &n);
  printf("Enter BT:\n");

  for (i = 0; i < n; i++) {
    printf("P%d: ", i + 1);
    scanf("%d", &A[i][1]);
    A[i][0] = i + 1;
  }
  // Sorting process according to their BT
  for (i = 0; i < n; i++) {
    index = i;
    for (j = i + 1; j < n; j++)
      if (A[j][1] < A[index][1])
        index = j;
    temp = A[i][1];
    A[i][1] = A[index][1];
    A[index][1] = temp;

    temp = A[i][0];
    A[i][0] = A[index][0];
    A[index][0] = temp;
  }
  A[0][2] = 0;
  // Calculation of Waiting Times
  for (i = 1; i < n; i++) {
    A[i][2] = 0;
    for (j = 0; j < i; j++)
```

```
      A[i][2] += A[j][1];
    total += A[i][2];
  }
  avg_wt = (float)total / n;
  total = 0;
  printf("P  BT  WT  TAT\n");
  // Calculate TAT
  for (i = 0; i < n; i++) {
    A[i][3] = A[i][1] + A[i][2];
    total += A[i][3];
    printf("P%d  %d  %d  %d\n", A[i][0],
      A[i][1], A[i][2], A[i][3]);
  }
  avg_tat = (float)total / n;
  printf("Average WT: %f", avg_wt);
  printf("\nAverage TAT: %f", avg_tat);
}
```

Listing 2. SJF Example

IV. Innovative Scheduling Algorithm: NOVEL

Since the FCFS and SJF Scheduling algorithms mainly take consideration on the arrival sequence and quantity number rather than the overall utilization of the three plants, which is the only judgement on PLS task, our group aims to design an innovative algorithm to reach better utilization. The design details, considerations and analysis are discussed in this section.

A. *Analysis on brute-force algorithm*

It is not difficult to find brute-force algorithm always generate the schedule with best utilization by comparing each possible schedules. However, it is not recommended for its time complexity and naïve idea without innovation. Even though some improvements could be made during comparation, the time complexity still requires O(2n * 3n * n!) for n orders in the worst case. In addition, because of its lack of analysis on the real problems, this scheduling suffers from its lack of creativity. Therefore, we recommend an algorithm using greedy idea and a series of improvements out of real needs.

B. *Reanalysis on PLS task*

Before going into details of our algorithm, a deeper look onto the PLS task is recommended, where you could discover the design considerations. For one, we regard a schedule as two components that is fragment and production, based on the states of the three plants idle and active. Therefore, to improve the overall utilization given by a specific period only needs to decrease the number of fragments. Take a closer look into the fragments, we divided them into two types: internal fragments and external fragments. Internal fragments refer to the idle state of a plant for one day, which is cause by the assigned production quantity is less than the capacity of the plant of one day. While the external fragments mean the whole idle state of one day with respect to a factory, which is caused by rejected orders could not be finished by that plant before order due date. In our design, we aim to lower both internal fragments and external fragments. For another, we focus on the numerator and denominator of the utilization formula. The denominator of utilization is unchanged when comparing different algorithms based on the same order set and production period. While the numerator is indeed the total sum of accepted orders. In this point of view, utilization improvement under same input (orders and period) is nothing but accept as many as possible orders.

C. *Design and Implementation Details*

On the one hand, fragmentations are decreased by using a combination of greedy and brute-force idea. More precisely, we design a macro scheduler aiming to lower external fragments and a micro scheduler which makes sure reach the least internal fragments. After given all the orders, the macro scheduler will sort the orders again in a descending quantity order then do the same thing as FCFS. This is reasonable to decrease external fragmentations because we believe the utilization of assigning some relatively large orders then filled with smaller orders is better than assigning relatively small orders but leave large orders rejected. Though this assumption may make mistakes under specific order batch, we design a performance test over randomly generated order quantity and due date, which proves this scheduling outperforms than FCFS and SJF. In addition, the macro scheduler decides the decline of orders, which is that the scheduler will exam if the current waiting order could be finished by three plants before its due days. We won't reject orders if they could be done without reserving space for the coming orders because we want to make sure every order assignment will perform best under current condition. This is also the drawback of the greedy idea that couldn't promise the optimal scheduling. After deciding the accepted order, the macro scheduler will ask the micro scheduler to decide how to allocate on three plants. At last, the macro scheduler will update the plant states based on the micro scheduler and move to the next order. On the other hand, the micro scheduler is given the remaining days of X, Y, Z plants before due date and total quantity the order requires. The micro scheduler will use brute-force design to compare all the distribution and decide the optimal one with least internal fragmentations. Then the allocation details will be sent back to the macro scheduler. At this point, the external and internal fragmentations is reduced in an optimized way. Last but not least, as we try to accept as many as orders as we could, we redesign the enumerate sequence and comparation conditions inside the micro scheduler and make sure if there are multiple schedule of the same least internal fragmentations, the final allocation will first use the plant X to remaining plant Z with larger daily capacity for accommodating more future orders. The pseudocode of the macro and micro scheduler is shown below for efficient understanding.

D. *Deficiency and Improvements*

Since greedy algorithm is applied, the NOVEL algorithm cannot promise to produce the optimal schedule, or even worse, produce the schedule worse than FCFS or SJF. Therefore, a checking mechanism is introduced in macro scheduler, which will compare the schedule with other two scheduling algorithms before sending back to other modules in pipe(). The final schedule with the best overall utilization will be sent

Algorithm 1 Macro Scheduler of NOVEL

reorder the orders in a descending quantity order
for each order in order_set **do**
  Calculate the remain days of X, Y, Z plants before due
  Calculate the capacity
  if $capacity < Q$ **then**
    reject the order
    break
  end if
  micro_scheduler(X_rem, Y_rem, Z_rem, Q)
  update the X, Y, Z plants states
end for

Algorithm 2 Micro Scheduler Function

function MICRO_SCHEDULER(X_rem, Y_rem, Z_rem, Q)
  $vac \leftarrow Q$
  $x \leftarrow 0$
  $y \leftarrow 0$
  $z \leftarrow 0$
  for $i \leftarrow 0$ **to** $\lfloor Q/300 \rfloor$ **do**
    for $j \leftarrow 0$ **to** $\lfloor Q/400 \rfloor$ **do**
      for $k \leftarrow 0$ **to** $\lfloor Q/500 \rfloor$ **do**
        $rem \leftarrow Q - 300 \cdot i - 400 \cdot j - 500 \cdot k$
        if $rem \geq 0$ **and** $rem \leq vac$ **then**
          $x \leftarrow i$
          $y \leftarrow j$
          $z \leftarrow k$
          $vac \leftarrow rem$
        end if
      end for
    end for
  end for
  return $\{x, y, z, vac\}$
end function

back to enhance performance. The performance experiments in Section eight provides a more straightforward optimization on NOVEL algorithm over the other two scheduling.

V. Software Structure of System

A three-layer system design is introduced to improve the modularity and clarity of the whole system and interprocess communications. The three layers is divided by their different functionality and processes, and this separation is not only a good realization of single responsibility design, but beneficial for our concurrent developments and tests that significantly improves efficiency. The first layer and the third layer are the parent process and serve as user interfaces of our system that are in charge of I/O modules and also the error handling modules. And the schedulers residents in the second layer, which is a separate forked process. The communications between different layers are done by the predefined data structure Order, Schedule and Report, which is transmitted through the unnamed pipes. The details will be unfolded as below, and the illustration is provided.

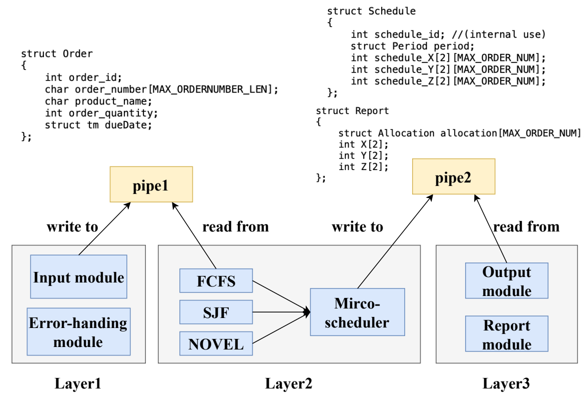


Fig. 1. Example of a full-column width figure.

- **First Layer:** The first layer contains the Input module and error handling module, which will perform the UI and functionality as the requirement need: receive period, orders, batch of orders with due date checking and run the scheduler by using fork() system call that create a new process with respect to the required scheduling algorithm. The order list and period store in the data structure shared by the parent and child will the sent through an unnamed pipe.
- **Second Layer:** The second layer accommodates three scheduling algorithms and is the Scheduler module, which will receive order list and period from the p2c pipe, schedule the orders, and send back the schedule details Schedule and Report that is a shared data structure by child and parent through c2p pipe. The child process is the above-mentioned macro scheduler, and it will call another independent method called kernel which is the micro scheduler. Good encapsulation is achieved since the responsibility of the micro scheduler is that dividing a number (order quantity) into 3 types of frames (daily capacity of three plants) with limited numbers of frames (the remaining days with respect to different plants before due date), thus, there is no dependency of micro scheduler onto macro scheduler. Lastly, the schedule details are gathered in different forms: gathered by plants (sent via Schedule data structure) is for the Output module and gathered by orders (sent via Report data structure) is for Report module use.
- **Third Layer:** The third layer is a relatively simpler tasks that in charge of the output and report format after decoding the data sent from layer two.

And there is no need to consider the complexity of running different layers, we divide the layer concepts into actions with smaller granularity, and we believe this implementation is better for the Procedure Oriented Language like C. And the procedure transformation is shown is the figure.

VI. Correctness Testing Cases

A. Test Case Description

The purpose of this test case is to validate the order processing capability of the Production Line Scheduler (PLS), ensuring it correctly handles order acceptance, rejection, and scheduling based on specified criteria such as due dates and production capacity.

B. Test Order Batch File

- **Order 1:**
  `addORDER P0301 2024-06-02 1826 Product_A`
  Expected to test the system's ability to reject orders that cannot be completed by the plant before the specified due date.
- **Order 2:**
  `addORDER P0302 2024-06-03 1330 Product_B`
  Expected to test the system's capacity to accept orders and allocate resources within the permissible time frame.
- **Order 3:**
  `addORDER P0303 2024-06-04 1427 Product_C`
  Aimed at testing the system's functionality to ignore orders that exceed the set due date.

C. Processed Result

- **Order 1:**
  Product NO.0301 was rejected, as the production capabilities could not meet the tight deadline. This confirms the system's functionality in evaluating and rejecting unfeasible production requests based on current plant capacities and due dates.
- **Order 2:**
  Product NO.0302 was accepted, and the system scheduled two days for completion. This showcases the system's effective scheduling and resource allocation capabilities, ensuring that feasible orders are processed efficiently.
- **Order 3:**
  Product NO.0303 was ignored because it exceeded the due date of "2024-06-03". This indicates the system's adherence to operational constraints and its ability to enforce order deadlines strictly.

Output Screen:

Report:

```
        ~~WELCOME TO PLS~~

Please enter:
> addPERIOD 2024-06-01 2024-06-03
Please enter:
> addBATCH orderBATCH16.dat
Some due dates are out of period, deemed invalid input, saved these lines to file InvalidInputs.txt.
Please enter:
> runPLS FCFS | printREPORT > report_01_FCFS.txt
=====================================================================================================
Plant_X (300 per day)
2024-06-01 to 2024-06-03

          Date    Product Name    Order Number   Quantity (Produced)    Due Date
=====================================================================================================
    2024-06-01       Product_B           P0302                  300  2024-06-03
    2024-06-02       Product_B           P0302                  230  2024-06-03
    2024-06-03              NA

=====================================================================================================

Plant_Y (400 per day)
2024-06-01 to 2024-06-03

          Date    Product Name    Order Number   Quantity (Produced)    Due Date
=====================================================================================================
    2024-06-01       Product_B           P0302                  400  2024-06-03
    2024-06-02       Product_B           P0302                  400  2024-06-03
    2024-06-03              NA

=====================================================================================================

Plant_Z (500 per day)
2024-06-01 to 2024-06-03

          Date    Product Name    Order Number   Quantity (Produced)    Due Date
=====================================================================================================
    2024-06-01              NA
    2024-06-02              NA
    2024-06-03              NA

=====================================================================================================
```

Fig. 2. This is the output on the terminal.

```
  UW PICO 5.09                                                              File: report_01_FCFS.txt

***PLS Schedule Analysis Report***

Algorithm used: FCFS

There are 1 Orders ACCEPTED. Details are as follows:

    ORDER NUMBER         START           END    DAYS    QUANTITY    PLANT
======================================================================
           P0302    2024-06-01    2024-06-02       2         530  Plant_X
           P0302    2024-06-01    2024-06-02       2         800  Plant_Y
- End -

======================================================================


There are 1 Orders REJECTED. Details are as follows:

    ORDER NUMBER    PRODUCT      DUE DATE    QUANTITY
=====================================================
           P0301  Product_A    2024-06-02        1826
- End -

=====================================================



***PERFORMANCE

Plant_X
       Number of days in use:                      2 days
       Number of products produced:              530 (in total)
       Utilization of the plant:               58.89 %

Plant_Y
       Number of days in use:                      2 days
       Number of products produced:              800 (in total)
       Utilization of the plant:               66.67 %

Plant_Z
       Number of days in use:                      0 days
       Number of products produced:                0 (in total)
       Utilization of the plant:                0.00 %

Overall of utilization:                        36.94 %
```

Fig. 3. This is the report.

D. Outcome Analysis

The outcomes from the PLS align with the expected results, demonstrating the system's capabilities in:

- **Adhering to Production Deadlines:**
  By rejecting orders that cannot be completed within the set deadlines, the system ensures operational efficiency and prevents overcommitment.
- **Resource Allocation:**
  Accepting and completing available orders within the designated timeframe shows effective resource management.
- **Enforcing Algorithm Rules:**
  Ignoring orders that exceed the due date highlights the system's strict compliance with algorithm policies.

E. Conclusion

These test orders confirm that the PLS operates correctly by adhering to algorithm rules. The system's ability to distinguish between feasible and infeasible orders, based on real-time data and predefined rules, ensures that production processes are both realistic and optimized. This testing phase not only validates the system's functional requirements but also reassures stakeholders of its reliability and efficiency in a live production environment.

VII. Performance Analysis

We implement the micro scheduler for all three scheduling algorithms. Even through the micro scheduler is one of our innovative components tailored for the NOVEL scheduling, we apply it onto the other two algorithms to improve their utilization performances. It is reasonable because the naïve FCFS scheduling only consider the arrival sequence rather than the overall utilization. Therefore, we design the same efficient kernel for these three schedulers. But still, the FCFS and SJF seldomly outperform NOVEL scheduling. As mentioned in last section, we believe that the utilization of assigning some relatively large orders then filled with smaller orders is better than assigning relatively small orders but leave large orders rejected. Therefore, the NOVEL algorithm is always better than the SJF scheduling as the running results show. While there is no relationship between the arrival sequence and quantity in our randomly generated data, so FCFS may be better than the NOVEL algorithm.

VIII. Program Set-up and Execution

This section provides detailed instructions on how to compile and execute the PLS (Product Line Scheduler) and discusses the Linux server environment used for testing and the results obtained.

A. Compilation Instructions

- **Prerequisites:**
  Ensure you have the necessary development tools installed. For a C/C++ project, you might need gcc or g++, and be sure that they are installed on the computer or server.
- **Clone the Repository:**

```
git clone https://github.com/hi
teacherIamhumble/PLS.git
cd PLS
```

- **Compile the Project:**

```
gcc PLS.c -std=c99 -o PLS
```

- **Run the Application:**

```
./PLS
```

- **See the Generated Files:**

```
pico orderBATCHXX.dat
pico report_XX_XXXX.dat
```

B. Library Required and Usage

We include a series of libraries of C which provide the functionality of using pipes, creating and using child process, using time data structure and conveniently handling string data type. In the below table, all the header files of our programs and the reasons they are used.

TABLE I
SPECIAL LIBRARIES IN USE TO SUPPORT SYSTEM

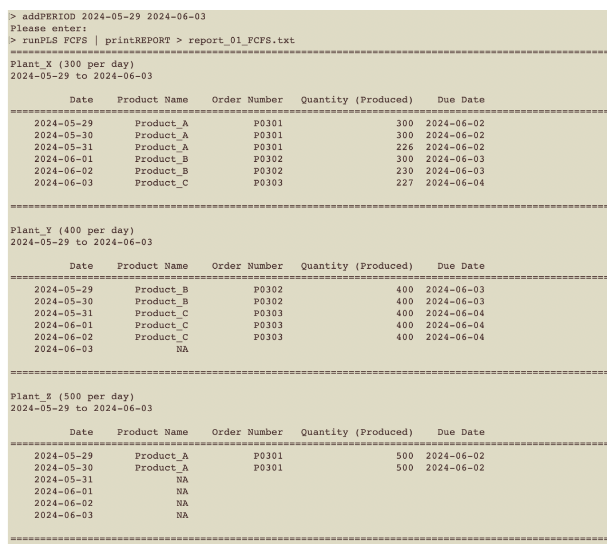
Library Used	Provided Functionality
<stdlib.h>	Provide the exit() function to processes
<sys/stat.h>	Provide the state of a child process back to its parent process
<sys/wait.h>	Provide waitpid() for a parent to ensure a specific child ends successfully
<unistd.h>	Provide functions that create pipes, close ends, and write/read from pipes.
<fcntl.h>	
<string.h>	Provide strcmp() to handle string type
<time.h>	Provide data structure tm to examine the correctness of input date and calculate the differences between two dates

C. Linux Server Testing Environment

We use the c99 rather than the default version of gcc compiler: c90. Below is the basic information of the COMP appolo server that we run tests on.

D. Test Results

Test cases were executed to verify the scheduling and order handling capabilities of the PLS. The tests included scenarios like order acceptance, rejection based on capacity, and deadline adherence.



```
> addPERIOD 2024-05-29 2024-06-03
Please enter:
> runPLS FCFS | printREPORT > report_01_FCFS.txt
========================================================================================
Plant_X (300 per day)
2024-05-29 to 2024-06-03

        Date    Product Name    Order Number   Quantity (Produced)    Due Date
========================================================================================
    2024-05-29      Product_A          P0301                   300  2024-06-02
    2024-05-30      Product_A          P0301                   300  2024-06-02
    2024-05-31      Product_A          P0301                   226  2024-06-02
    2024-06-01      Product_B          P0302                   300  2024-06-03
    2024-06-02      Product_B          P0302                   230  2024-06-03
    2024-06-03      Product_C          P0303                   227  2024-06-04

========================================================================================

Plant_Y (400 per day)
2024-05-29 to 2024-06-03

        Date    Product Name    Order Number   Quantity (Produced)    Due Date
========================================================================================
    2024-05-29      Product_B          P0302                   400  2024-06-03
    2024-05-30      Product_B          P0302                   400  2024-06-03
    2024-05-31      Product_C          P0303                   400  2024-06-04
    2024-06-01      Product_C          P0303                   400  2024-06-04
    2024-06-02      Product_C          P0303                   400  2024-06-04
    2024-06-03             NA

========================================================================================

Plant_Z (500 per day)
2024-05-29 to 2024-06-03

        Date    Product Name    Order Number   Quantity (Produced)    Due Date
========================================================================================
    2024-05-29      Product_A          P0301                   500  2024-06-02
    2024-05-30      Product_A          P0301                   500  2024-06-02
    2024-05-31             NA
    2024-06-01             NA
    2024-06-02             NA
    2024-06-03             NA

========================================================================================
```

Fig. 4. This is the output on the terminal

E. Conclusion

All test cases passed successfully. The scheduler was able to handle multiple concurrent orders and optimize the production line efficiently

IX. Result Discussion

We generate 5 random order batches by Microsoft excel and test them on the three scheduling algorithms. And the results are shown in the following table:

TABLE II
PERFORMANCE RESULTS

Batch no.	FCFS	SJF	NOVEL
1	62.49%	60.52%	69.69%
2	76.95%	71.24%	76.95%
3	78.25%	81.73%	85.77%
4	59.43%	59.43%	59.43%
5	55.99%	55.99%	61.71%

The table provides utilization percentages for different batches processed through three scheduling algorithms: First-Come First-Served (FCFS), Shortest Job First (SJF), and our NOVEL algorithm (NOVEL). Upon analysis, it's evident that the NOVEL algorithm consistently outperforms FCFS and SJF in terms of plant utilization across all batches. In Batch 1, NOVEL achieves a utilization rate of 69.69%, surpassing FCFS (62.49%) and SJF (60.52%). This trend continues in Batch 2 and Batch 3, where NOVEL maintains or exceeds the highest utilization percentages among the three algorithms. Particularly noteworthy is Batch 3, where NOVEL achieves an impressive 85.77% utilization rate compared to FCFS (78.25%) and SJF (81.73%). The superior performance of NOVEL can be attributed to its unique prioritization of orders based on larger quantities. By favoring high-volume orders, NOVEL optimizes throughput and resource allocation, thereby maximizing plant utilization. This is evident in the consistently higher utilization rates observed across all batches. Furthermore, the NOVEL algorithm demonstrates resilience in Batch 4, where all algorithms yield identical utilization rates (59.43%). While FCFS and SJF falter in adapting to the batch's characteristics, NOVEL maintains its effectiveness by efficiently handling orders with larger quantities. In Batch 5, NOVEL continues to exhibit its superiority, achieving a utilization rate of 61.71% compared to FCFS and SJF, both at 55.99%. This further reinforces the efficacy of the NOVEL algorithm in dynamically managing production schedules and optimizing resource utilization. Overall, the results highlight the significant advantages of the NOVEL algorithm in enhancing plant utilization and optimizing production schedules in a real-world manufacturing context. Its ability to adapt to varying batch characteristics and prioritize high-volume orders underscores its potential for driving efficiency and productivity in industrial settings.

X. CONCLUSION

In this steel-making plant problem, we made great efforts to solve the scheduling inefficiencies. By incorporating CPU scheduling algorithm ideas into the structure of our production line scheduler (PLS), we've accomplished better resource allocation and overall system performance. Through development and implementation of our program in a Linux environment, we've created a dynamic platform which is capable of adjusting to real-time needs and requirements. This versatility means that our scheduler can respond to changing production demands, reducing delays, and increasing throughput across numerous sites.

Besides, the use of traditional CPU scheduling methods such as First-Come First-Served (FCFS) and Shortest Job First (SJF), combined with our new strategy that prioritizes greater volumes, has provided significant inspirations of the intricacies of production scheduling. We investigated how each algorithm operates under different settings, which reveals their relative strengths and drawbacks in the industrial environment. Furthermore, our project components' structural was close to essential operating system features, such as work queues and system monitoring tools, and it allows for the combination of theoretical concepts and practical applications. This consistency not only accelerates the development process, but also establishes a conceptual framework for future improvements and revisions.

In conclusion, our performance analysis has revealed actionable knowledge for the medium-size steel-making manufacturer, allowing him to make more smart decisions and prepare strategically. The manufacturer may use the data supplied by our scheduler to improve their operations, eliminate idle time, and eventually become more competitive in the industry. In essence, our project applies computer science insights to real-world manufacturing difficulties. As we continue to improve and extend our scheduling, we are dedicated to fostering innovation and efficiency in the ever-changing face of industrial production.

XI. REFERENCES

REFERENCES

[1] YA Adekunle, ZO Ogunwobi, A Sarumi Jerry, BT Efuwape, Seun Ebiesuwa, and Jean-Paul Ainam. A comparative study of scheduling algorithms for multiprogramming in real-time systems. *International Journal of Innovation and Scientific Research*, 12(1):180–185, 2014.

[2] Ali A AL-Bakhrani, Abdulnaser A Hagar, Ahmed A Hamoud, and S Kawathekar. Comparative analysis of cpu scheduling algorithms: Simulation and its applications. *International Journal of Advanced Science and Technology*, 29(3):483–494, 2020.

[3] Sajida Fayyaz, Hafiz Ali Hamza, Saria Moin U Din, and Ehatsham Riaz. Comparative analysis of basic cpu scheduling algorithms. *International Journal of Multidisciplinary Sciences and Engineering*, 8, 2017.

[4] Neetu Goel and RB Garg. A comparative study of cpu scheduling algorithms. *arXiv preprint arXiv:1307.4165*, 2013.

[5] Kenneth L Krause, Vincent Y Shen, and Herbert D Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *Journal of the ACM (JACM)*, 22(4):522–550, 1975.

[6] Ishwari Singh Rajput and Deepa Gupta. A priority based round robin cpu scheduling algorithm for real time systems. *International Journal of Innovations in Engineering and Technology*, 1(3):1–11, 2012.

[7] Abraham Silberschatz, James L Peterson, and Peter B Galvin. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[8] Ajit Singh, Priyanka Goyal, and Sahil Batra. An optimized round robin scheduling algorithm for cpu scheduling. *International Journal on Computer Science and Engineering*, 2(07):2383–2385, 2010.

XII. INNOVATIVE SCHEDULING ALGORITHM: NOVEL

Since the FCFS and SJF Scheduling algorithms mainly take consideration on the arrival sequence and quantity number rather than the overall utilization of the three plants, which is the only judgement on PLS task, our group aims to design an innovative algorithm to reach better

utilization. The design details, considerations and analysis are discussed in this section.

A. Analysis on brute-force algorithm

It is not difficult to find brute-force algorithm always generate the schedule with best utilization by comparing each possible schedules. However, it is not recommended for its time complexity and naïve idea without innovation. Even though some improvements could be made during comparation, the time complexity still requires O(2n * 3n * n!) for n orders in the worst case. In addition, because of its lack of analysis on the real problems, this scheduling suffers from its lack of creativity. Therefore, we recommend an algorithm using greedy idea and a series of improvements out of real needs.

B. Reanalysis on PLS task

Before going into details of our algorithm, a deeper look onto the PLS task is recommended, where you could discover the design considerations. For one, we regard a schedule as two components that is fragment and production, based on the states of the three plants idle and active. Therefore, to improve the overall utilization given by a specific period only needs to decrease the number of fragments. Take a closer look into the fragments, we divided them into two types: internal fragments and external fragments. Internal fragments refer to the idle state of a plant for one day, which is cause by the assigned production quantity is less than the capacity of the plant of one day. While the external fragments mean the whole idle state of one day with respect to a factory, which is caused by rejected orders could not be finished by that plant before order due date. In our design, we aim to lower both internal fragments and external fragments. For another, we focus on the numerator and denominator of the utilization formula. The denominator of utilization is unchanged when comparing different algorithms based on the same order set and production period. While the numerator is indeed the total sum of accepted orders. In this point of view, utilization improvement under same input (orders and period) is nothing but accept as many as possible orders.

C. Design and Implementation Details

On the one hand, fragmentations are decreased by using a combination of greedy and brute-force idea. More precisely, we design a macro scheduler aiming to lower external fragments and a micro scheduler which makes sure reach the least internal fragments. After given all the orders, the macro scheduler will sort the orders again in a descending quantity order then do the same thing as FCFS. This is reasonable to decrease external fragmentations because we believe the utilization of assigning some relatively large orders then filled with smaller orders is better than assigning relatively small orders but leave large orders rejected. Though this assumption may make mistakes under specific order batch, we design a performance test over randomly generated order quantity and due date, which proves this scheduling outperforms than FCFS and SJF. In addition, the macro scheduler decides the decline of orders, which is that the scheduler will exam if the current waiting order could be finished by three plants before its due days. We won't reject orders if they could be done without reserving space for the coming orders because we want to make sure every order assignment will perform best under current condition. This is also the drawback of the greedy idea that couldn't promise the optimal scheduling. After deciding the accepted order, the macro scheduler will ask the micro scheduler to decide how to allocate on three plants. At last, the macro scheduler will update the plant states based on the micro scheduler and move to the next order. On the other hand, the micro scheduler is given the remaining days of X, Y, Z plants before due date and total quantity the order requires. The micro scheduler will use brute-force design to compare all the distribution and decide the optimal one with least internal fragmentations. Then the allocation details will be sent back to the macro scheduler. At this point, the external and internal fragmentations is reduced in an optimized way. Last but not least, as we try to accept as many as orders as we could, we redesign the enumerate sequence and comparation conditions inside the micro scheduler and make sure if there are multiple schedule of the same least internal fragmentations, the final allocation will first use the plant X to remaining plant Z with larger daily capacity for accommodating more future orders. The pseudocode of the macro and micro scheduler is shown below for efficient understanding.

Algorithm 1 Macro Scheduler of NOVEL

reorder the orders in a descending quantity order
for each order in order_set **do**
  Calculate the remain days of X, Y, Z plants before due
  Calculate the capacity
  if $capacity < Q$ **then**
    reject the order
    break
  end if
  micro_scheduler(X_rem, Y_rem, Z_rem, Q)
  update the X, Y, Z plants states
end for

D. Deficiency and Improvements

Since greedy algorithm is applied, the NOVEL algorithm cannot promise to produce the optimal schedule, or even worse, produce the schedule worse than FCFS or SJF. Therefore, a checking mechanism is introduced in macro scheduler, which will compare the schedule with other two scheduling algorithms before sending back to other modules in pipe(). The final schedule with the best overall utilization will be sent back to enhance performance. The

Algorithm 4 Micro Scheduler Function

function MICRO_SCHEDULER(X_rem, Y_rem, Z_rem, Q)
 $vac \leftarrow Q$
 $x \leftarrow 0$
 $y \leftarrow 0$
 $z \leftarrow 0$
 for $i \leftarrow 0$ **to** $\lfloor Q/300 \rfloor$ **do**
  for $j \leftarrow 0$ **to** $\lfloor Q/400 \rfloor$ **do**
   for $k \leftarrow 0$ **to** $\lfloor Q/500 \rfloor$ **do**
    $rem \leftarrow Q - 300 \cdot i - 400 \cdot j - 500 \cdot k$
    if $rem \geq 0$ **and** $rem \leq vac$ **then**
     $x \leftarrow i$
     $y \leftarrow j$
     $z \leftarrow k$
     $vac \leftarrow rem$
    end if
   end for
  end for
 end for
 return $\{x, y, z, vac\}$
end function

performance experiments in Section eight provides a more straightforward optimization on NOVEL algorithm over the other two scheduling.

XIII. Appendix

A. Source Code

```c
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>


#define MAX_DATE_LEN 32
struct tm startDate = {0}, endDate = {0};
char startDateStr[MAX_DATE_LEN], endDateStr[MAX_DATE_LEN]; // start date and end date for the period
#define MAX_ALGORITHMNAME_LEN 20
char algorithmName[MAX_ALGORITHMNAME_LEN];// name of the current algorithm for schedule module
#define MAX_FILENAME_LEN 50
char reportFileName[MAX_FILENAME_LEN]; // name of the report file for analysis report

// when invoking a scheduler, pass the period
struct Period
{
    struct tm startDate;
    struct tm endDate;
};
// struct Period currentPeriod = {{0, 0, 0}, {0, 0, 0}};addPERIOD 2024-05-01 2024-06-25

#define MAX_ORDERNUMBER_LEN 10
struct Order
{
    int order_id;                              // arrival sequence(internal use)
    char order_number[MAX_ORDERNUMBER_LEN]; // order numbeR
    char product_name;                         // 1 of 9 letters: A, B, C, D, E, F, G, H, I
    int order_quantity;                        // quantity of product
    struct tm dueDate;                         // due date
};
#define MAX_ORDER_NUM 200
int orderNum = 0;
struct Order order[MAX_ORDER_NUM];

struct Allocation
{
    // one of this struct shows the allocation of one order
    int order_id; // order id, initialized to -1
    int accepted; // 1 for accepted, 0 for denied, initialized to -1
    // first dimension is plant x at 0, plant y at 1, plant z at 2
    // second demension is days since start date at 0, days to produce in that plant at 1, quantity at 2
    // initialized to -1
    int schedule[3][3]; //plant_id: days since start date, days to produce in that plant, quantity
};

struct Schedule
{
    int schedule_id; //(internal use)
    struct Period period;
    // first dimension is order id at 0 and quantity at 1, initialized to -1
    // second dimension is the day starting from 0 which is the start date
    // if not job that day, order id is -1, quantity is not specified
    int schedule_X[2][MAX_ORDER_NUM]; // order_id, quantity
    int schedule_Y[2][MAX_ORDER_NUM]; // order_id, quantity
    int schedule_Z[2][MAX_ORDER_NUM]; // order_id, quantity
};

struct Report
{
    // this is uesd in the analysis report
    // for each order, there is a allocation struct
    // assume that the order id is the same as the index in the array
    struct Allocation allocation[MAX_ORDER_NUM];
    // number of days in use at 0, number of quantity produced at 1, initialized to -1
    int X[2]; // days, quantity
    int Y[2]; // days, quantity
    int Z[2]; // days, quantity
};

// this is to contain the infos about communication between parent and child process
struct Scheduler
{
    pid_t pid;
    int fdp2c[2]; // pipe from parent to child
    int fdc2p[2]; // pipe from child to parent
};

void promptEnter();
```

```c
int startsWith(const char *str, const char *prefix);
const struct tm str2Date(const char *str);
void addPeriod(const char *str);
int addOrder(const char *str);
void addBatch(const char *str);
void runPLS(const char *str);
void exitPLS();
void work(const char *algorithm, const char *filename);
void parseInput(const char *str);
void inputModule();
int dateDiff(const struct tm *startDate, const struct tm *endDate);
void writeReport(struct Report *report);

int kernel(int Q, int Rx, int Ry, int Rz, int alloc[3]) {

    int delta = Q;

    for (int i = 0; i <= Rx; ++i) {
        for (int j = 0; j <= Ry; ++j) {
            for (int k = 0; k <= Rz; ++k) {
                int remain = i * 300 + j * 400 + k * 500 - Q;
                if (remain >= 0 && remain <= delta) {
                    alloc[0] = i;
                    alloc[1] = j;
                    alloc[2] = k;
                    delta = remain;
                }
            }
        }
    }

    int re = alloc[0] * 300 + alloc[1] * 400 + alloc[2] * 500 - Q;
    //find which plant where the vacancy from
    if (re == 0) {
        return 0;
    } else if (re >= 400 && alloc[2] > 0) {
        return 3;
    } else if (re >=300 && alloc[1] > 0) {
        return 2;
    } else if (re >= 300 && alloc[2] > 0) {
        return 3;
    } else if (alloc[0] > 0) {
        return 1;
    } else if (alloc[1] > 0) {
        return 2;
    } else if (alloc[2] > 0) {
        return 3;
    } else {
        return -1;
    }
}
void promptEnter()
{
    printf("Please enter:\n> ");
}
int startsWith(const char *str, const char *prefix)
{
    if (str == NULL || prefix == NULL)
        return 0; // Handle NULL pointers
    size_t len_prefix = strlen(prefix);
    size_t len_str = strlen(str);

    if (len_prefix > len_str)
        return 0; // Prefix longer than string cannot be a prefix

    return (strncmp(str, prefix, len_prefix) == 0);
}
```

```c
// const struct Date str2Date(const char *str){
//     struct Date res;
//     sscanf(str, "%d-%d-%d", &res.year, &res.month, &res.day);
//     return res;
// }
const struct tm str2Date(const char *str)
{
    int year, month, day;
    sscanf(str, "%d-%d-%d", &year, &month, &day);
    struct tm res = {0};
    res.tm_year = year - 1900; // year 1900 being 0
    res.tm_mon = month - 1;    // January being 0
    res.tm_mday = day;         // day of the month, day 1 being 1
    return res;
}

int dateDiff(const struct tm *startDate, const struct tm *endDate)
{
    time_t start = mktime((struct tm *)startDate);
    time_t end = mktime((struct tm *)endDate);
    if (start == -1 || end == -1)
    {
        perror("mktime");
        exit(1);
    }
    double diff = difftime(end, start);
    return (int)diff / (60 * 60 * 24);
}


// const char* date2Str(const struct Date date){
//     char res[MAX_DATE_LEN];
//     sprintf(res, "%d-%d-%d", date.year, date.month, date.day);
//     // printf("result is %s\n", res);
//     return res;
// }
void addPeriod(const char *str)
{
    // str is a addPERIOD command.
    // char startDateStr[MAX_DATE_LEN];
    // char endDateStr[MAX_DATE_LEN];
    sscanf(str, "addPERIOD %s %s", startDateStr, endDateStr);
    // printf("start date is %s\n", startDateStr);
    // printf("end date is %s\n", endDateStr);
    startDate = str2Date(startDateStr);
    endDate = str2Date(endDateStr);
    // printf("start date is %d-%d-%d\n", startDate.year, startDate.month, startDate.day);
    // printf("start date is %d-%d-%d\n", endDate.year, endDate.month, endDate.day);
}
const char INVALID_INPUTS[] = "InvalidInputs.txt";
void appendToInvalidFile(const char *str){
    FILE *filePtr;
    // Open the file in append mode
    filePtr = fopen(INVALID_INPUTS, "a");
    if (filePtr == NULL) {
        perror("Error opening file");
        return;
    }

    // Append the line to the file
    if (fprintf(filePtr, "%s\n", str) < 0) {
        perror("Error writing to file");
        // Close the file before returning
        fclose(filePtr);
        return;
    }

```

```c
    // Close the file
    fclose(filePtr);
}
int CheckDueDate(struct tm orderDueDate){
    // check if that due date is within the period
    // if is, return 1
    // if not, return 0
    if (dateDiff(&startDate, &orderDueDate) < 0 || dateDiff(&endDate, &orderDueDate) > 0)
    {
        return 0;
    }
    return 1;

}
int addOrder(const char *str)
{
    // str is a addORDER command.
    char dueDateStr[MAX_DATE_LEN];
    sscanf(str, "addORDER %s %s %d Product_%c", order[orderNum].order_number, dueDateStr, &order[orderNum].order_quantity, &order[orderNum].product_name);
    struct tm orderDueDate = str2Date(dueDateStr);
    int n = CheckDueDate(orderDueDate);
    if ( n == 0 ){
        appendToInvalidFile(str);
        return 1;
    }
    order[orderNum].order_id = orderNum;
    order[orderNum].dueDate = str2Date(dueDateStr);
    ++orderNum;
    return 0;
}
void addBatch(const char *str)
{
    char filename[MAX_FILENAME_LEN];
    sscanf(str, "addBATCH %s", filename);
    FILE *file = fopen(filename, "r");
    if (file == NULL)
    {
        perror("Failed to open file");
        exit(1);
    }
    const int MAX_INPUT_LEN = 1024;
    char buffer[MAX_INPUT_LEN];
    int invalid = 0;
    while (fgets(buffer, sizeof(buffer), file) != NULL)
    {
        // Remove newline character if present
        buffer[strcspn(buffer, "\n")] = 0;
        //printf("You entered: %s\n", buffer);
        int p = addOrder(buffer);
        if (p == 1) {
            invalid = 1;
        }
    }
    if (invalid == 1) {
       printf("Some due dates are out of period, deemed invalid input, saved these lines to file InvalidInputs.txt.\n");
    }
}

void runPLS(const char *str)
{
    sscanf(str, "runPLS %s | printREPORT > %s", algorithmName, reportFileName);
    //printf("use algorithm %s, report file to %s\n", algorithmName, reportFileName);
    work(algorithmName, reportFileName);
}
void exitPLS()
{
    printf("Bye-bye!");
    exit(0);
}
void parseInput(const char *str)
{
    if (startsWith(str, "addPERIOD"))
    {
        //puts("This is a addPERIOD command.");
        addPeriod(str);
    }
    else if (startsWith(str, "addORDER"))
    {
        //puts("This is a addORDER command.");
        addOrder(str);
    }
    else if (startsWith(str, "addBATCH"))
    {
        //puts("This is a addBATCH command.");
        addBatch(str);
    }
    else if (startsWith(str, "runPLS"))
    {
        //puts("This is a runPLS command.");
        runPLS(str);
    }
    else if (startsWith(str, "exitPLS"))
    {
        //puts("This is a exitPLS command.");
        exitPLS();
    }
    else
    {
        fprintf(stderr, "Error: Input command invalid format.\n");
        exit(1);
    }
}
void inputModule()
{
    printf("\n\t~~WELCOME TO PLS~~\n\n");
    while (1)
    {
        promptEnter();
        const int MAX_INPUT_LEN = 1024;
        char buffer[MAX_INPUT_LEN];
        if (fgets(buffer, sizeof(buffer), stdin))
        {
            // Remove newline character if present
            buffer[strcspn(buffer, "\n")] = 0;
            //("You entered: %s\n", buffer);
            parseInput(buffer);
        }
    }
}
struct tm calcDate(struct tm date, int offset){
    //offset is the number of days to add to date
    //result is the date after adding offset days
    struct tm result = date;
    result.tm_mday += offset;
    mktime(&result);
    return result;
}
void dateToStr(struct tm date, char *str){
    //convert date to string
    strftime(str, MAX_DATE_LEN, "%Y-%m-%d", &date);
}
int diffDate(struct tm date1, struct tm date2){
    //date1 is the earlier date
    //date2 is the later date
    //return the difference in days between date1 and date2
    int diff = dateDiff(&date1, &date2);
    return diff;
```

```c
}
void printSchedule(struct Schedule schedule){
    //first print plant x
    printf("===============================\n");
    printf("Plant_X (300 per day)\n");
    printf("%s to %s\n", startDateStr, endDateStr);
    printf("\n");
    //print name of each column
    //first column has a width of 14 character
    //second column has a width of 16 character
    //third column has a width of 16 character
    //fourth column has a width of 22 character
    //fifth column has a width of 12 character
    printf("%14s%16s%16s%22s%12s\n", "Date", "Product Name", "Order Number", "Quantity (Produced)", "Due Date");
    printf("=============================\n");
    int totalDays = diffDate(startDate, endDate)+1; //including the start date
    for(int i = 0; i < totalDays; i++) {// go over each day
        char todayDateStr[MAX_DATE_LEN];
        struct tm todayDate = calcDate(startDate, i);
        dateToStr(todayDate, todayDateStr);
        if(schedule.schedule_X[0][i] == -1) {// no order on this day
            printf("%14s%16s\n", todayDateStr, "NA");
            continue;
        }
        int orderID = schedule.schedule_X[0][i];
        int quantity = schedule.schedule_X[1][i];
        char productName[10] = "Product_";
        char temp[2] = {order[orderID].product_name, '\0'};
        strcat(productName, temp);
        char ordreDueDateStr[MAX_DATE_LEN];
        dateToStr(order[orderID].dueDate, ordreDueDateStr);
        printf("%14s%16s%16s%22d%12s\n", todayDateStr, productName, order[orderID].order_number, quantity, ordreDueDateStr);
    }
    printf("\n");
    printf("=======================================\n");
    printf("\n");
    //then print plant y
    printf("Plant_Y (400 per day)\n");
    printf("%s to %s\n", startDateStr, endDateStr);
    printf("\n");
    printf("%14s%16s%16s%22s%12s\n", "Date", "Product Name", "Order Number", "Quantity (Produced)", "Due Date");
    printf("=======================================\n");
    for(int i = 0; i < totalDays; i++) {// go over each day
        char todayDateStr[MAX_DATE_LEN];
        struct tm todayDate = calcDate(startDate, i);
        dateToStr(todayDate, todayDateStr);
        if(schedule.schedule_Y[0][i] == -1) {// no order on this day
            printf("%14s%16s\n", todayDateStr, "NA");
            continue;
        }
        int orderID = schedule.schedule_Y[0][i];
        int quantity = schedule.schedule_Y[1][i];
        char productName[10] = "Product_";
        char temp[2] = {order[orderID].product_name, '\0'};
        strcat(productName, temp);
        char ordreDueDateStr[MAX_DATE_LEN];
        dateToStr(order[orderID].dueDate, ordreDueDateStr);
        printf("%14s%16s%16s%22d%12s\n", todayDateStr, productName, order[orderID].order_number, quantity, ordreDueDateStr);
    }
    printf("\n");
    printf("===========================================\n");
    printf("\n");
    //then print plant z
    printf("Plant_Z (500 per day)\n");
    printf("%s to %s\n", startDateStr, endDateStr);
    printf("\n");
    printf("%14s%16s%16s%22s%12s\n", "Date", "Product Name", "Order Number", "Quantity (Produced)", "Due Date");
    printf("============================================\n");
    for(int i = 0; i < totalDays; i++) {// go over each day
        char todayDateStr[MAX_DATE_LEN];
        struct tm todayDate = calcDate(startDate, i);
        dateToStr(todayDate, todayDateStr);
        if(schedule.schedule_Z[0][i] == -1) {// no order on this day
            printf("%14s%16s\n", todayDateStr, "NA");
            continue;
        }
        int orderID = schedule.schedule_Z[0][i];
        int quantity = schedule.schedule_Z[1][i];
        char productName[10] = "Product_";
        char temp[2] = {order[orderID].product_name, '\0'};
        strcat(productName, temp);
        char ordreDueDateStr[MAX_DATE_LEN];
        dateToStr(order[orderID].dueDate, ordreDueDateStr);
        printf("%14s%16s%16s%22d%12s\n", todayDateStr, productName, order[orderID].order_number, quantity, ordreDueDateStr);
    }
    printf("\n");
    printf("==============================================\n");
    printf("\n");
}
void work(const char *algorithm, const char *filename)
{
    struct Scheduler scheduler;
    if (pipe(scheduler.fdp2c) < 0 || pipe(scheduler.fdc2p) < 0)
    {
        perror("pipe");
        exit(1);
    }
    scheduler.pid = fork();
    if (scheduler.pid < 0)
    {
        perror("error forking.");
        exit(1);
    }
    if (scheduler.pid == 0) //child process
    {
        close(scheduler.fdp2c[1]); // close write end of parent to child pipe
        close(scheduler.fdc2p[0]); // close read end of child to parent pipe
```

```c
    // read algorithm name from parent
    int messageLen;
    if (read(scheduler.fdp2c[0], &messageLen, sizeof(messageLen)) < 0)
    {
        perror("error when reading message length from parent");
        exit(EXIT_FAILURE);
    }
    char algorithmName[MAX_ALGORITHMNAME_LEN];
    int n = read(scheduler.fdp2c[0], algorithmName, messageLen);
    if (n < 0)
    {
        perror("error when reading algorithm name from parent");
        exit(EXIT_FAILURE);
    }
    // read period from parent
    struct Period period;
    n = read(scheduler.fdp2c[0], &period, sizeof(period));
    if (n < 0)
    {
        perror("error when reading period from parent");
        exit(EXIT_FAILURE);
    }
    // read orders from parent
    // number of orders to come
    int orderNum;
    if (read(scheduler.fdp2c[0], &orderNum, sizeof(orderNum)) < 0)
    {
        perror("error when reading orderNum from parent");
        exit(EXIT_FAILURE);
    }
    // orders
    struct Order order[3][orderNum];
    if (read(scheduler.fdp2c[0], order[0], sizeof(struct Order) * orderNum) < 0)
    {
        perror("error when reading orders from parent");
        exit(EXIT_FAILURE);
    }
    // make two order list copys
    for (int i = 0; i < orderNum; i++)
    {
        order[1][i] = order[0][i];
        order[2][i] = order[0][i];
    }
    // schedule & report
    // initialize schedule
    struct Schedule schedule[3];
    for (int m = 0; m < 3; ++ m) {
        schedule[m].schedule_id = 0;
        schedule[m].period = period;
         for (int i = 0; i < MAX_ORDER_NUM; i++)
        {
            schedule[m].schedule_X[0][i] = -1;
            schedule[m].schedule_X[1][i] = -1;
            schedule[m].schedule_Y[0][i] = -1;
            schedule[m].schedule_Y[1][i] = -1;
            schedule[m].schedule_Z[0][i] = -1;
            schedule[m].schedule_Z[1][i] = -1;
        }
    }


    // initialize report
    struct Report report[3];

    for (int m =0; m < 3; m++) {
        for (int i = 0; i < MAX_ORDER_NUM; i++) {
        report[m].allocation[i].order_id = -1;
        report[m].allocation[i].accepted = -1;
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                report[m].allocation[i].schedule[j][k] = -1;
                report[m].allocation[i].schedule[j][k] = -1;
                report[m].allocation[i].schedule[j][k] = -1;
            }
        }
        }
    }


    // SJF
    for (int i = 0; i < orderNum - 1; i++)
    {
        for (int j = 0; j < orderNum - i - 1; j++)
        {
            if (order[1][j].order_quantity > order[1][j + 1].order_quantity)
            {
                struct Order temp = order[1][j];
                order[1][j] = order[1][j + 1];
                order[1][j + 1] = temp;
            }
        }
    }

    // NOVEL
    for (int i = 0; i < orderNum - 1; i++)
    {
        for (int j = 0; j < orderNum - i - 1; j++)
        {
            if (order[2][j].order_quantity < order[2][j + 1].order_quantity)
            {
                struct Order temp = order[2][j];
                order[2][j] = order[2][j + 1];
                order[2][j + 1] = temp;
            }
        }
    }

    for (int m = 0; m < 3; m ++) {
        // tranverse the order list to generate schedule
        int currentX = 0, currentY = 0, currentZ = 0; //current production days of X, Y, Z
        int X_remain = 0, Y_remain = 0, Z_remain = 0; //remaining days of X, Y, Z before due
        int X_total = 0, Y_total = 0, Z_total = 0;    //total quantity of X, Y, Z
        for (int i = 0; i < orderNum; i++)
        {
            int id = order[m][i].order_id;
            X_remain = dateDiff(&period.startDate, &order[m][i].dueDate) - currentX;
            Y_remain = dateDiff(&period.startDate, &order[m][i].dueDate) - currentY;
            Z_remain = dateDiff(&period.startDate, &order[m][i].dueDate) - currentZ;
            // Acceptance Judge
            // deney the order if the three plants cannot produce the product before its due
```

```
 date
            if (300 * X_remain + 400 * Y_remain + 500 * Z_remain < order[m][id].order_quantity)
            {
                // deney the order
                report[m].allocation[id].order_id = id;
                report[m].allocation[id].accepted = 0;
                continue;
            }
            X_remain = 0 > X_remain ? 0 : X_remain;
            Y_remain = 0 > Y_remain ? 0 : Y_remain;
            Z_remain = 0 > Z_remain ? 0 : Z_remain;
            // Allocation Calculation
            int alloc[3] = {0, 0, 0}; // days to assign to X, Y, Z
            //int vacancy = allocate(order[i].order_quantity, X_remain, Y_remain, Z_remain, alloc); // which plant has internal fragmentation
            int vacancy;
            vacancy = kernel(order[m][i].order_quantity, X_remain, Y_remain, Z_remain, alloc); // which plant has internal fragmentation


            if (vacancy == -1)
            {
                printf("error: invalid vacancy\n");
                exit(1);
            }

            // Report Generation
            // record the allocation for current order
            report[m].allocation[id].order_id = id;
            report[m].allocation[id].accepted = 1;
            report[m].allocation[id].schedule[0][0] = currentX + 1;
            report[m].allocation[id].schedule[1][0] = currentY + 1;
            report[m].allocation[id].schedule[2][0] = currentZ + 1;
            report[m].allocation[id].schedule[0][1] = alloc[0];
            report[m].allocation[id].schedule[1][1] = alloc[1];
            report[m].allocation[id].schedule[2][1] = alloc[2];
            report[m].allocation[id].schedule[0][2] = 0;
            report[m].allocation[id].schedule[1][2] = 0;
            report[m].allocation[id].schedule[2][2] = 0;

            // Schedule Generation
            for (int j = 0; j < alloc[0]; j++, ++currentX)
            {
                //Plant X: day[currentX] produce 300 quantity
                schedule[m].schedule_X[0][currentX] = id;
                schedule[m].schedule_X[1][currentX] = 300;
                //current order produced in plant X
                report[m].allocation[id].schedule[0][2] += 300;

            }
            for (int j = 0; j < alloc[1]; j++, ++currentY)
            {
                //Plant Y: day[currentY] produce 400 quantity
                schedule[m].schedule_Y[0][currentY] = id;
                schedule[m].schedule_Y[1][currentY] = 400;
                //current order produced in plant Y
                report[m].allocation[id].schedule[1][2] += 400;

            }
            for (int j = 0; j < alloc[2]; j++, ++currentZ)
            {
                //Plant Z: day[currentZ] produce 500 quantity
                schedule[m].schedule_Z[0][currentZ] = id;
                schedule[m].schedule_Z[1][currentZ] = 500;
                //current order produced in plant Z
                report[m].allocation[id].schedule[2][2] += 500;
            }
            // internal fragmentation handling
            int remain = (alloc[0] * 300 + alloc[1] * 400 + alloc[2] * 500) - order[m][i].order_quantity;
            switch(vacancy) {
                case 0: // no internal fragmentation
                    break;
                case 1: // internal fragmentation exists in X
                    schedule[m].schedule_X[1][currentX] = 300 - remain;
                    report[m].allocation[id].schedule[0][2] -= remain;
                    break;
                case 2:
                    schedule[m].schedule_Y[1][currentY] = 400 - remain;
                    report[m].allocation[id].schedule[1][2] -= remain;
                    break;
                case 3:
                    schedule[m].schedule_Z[1][currentZ] = 500 - remain;
                    report[m].allocation[id].schedule[2][2] -= remain;
                    break;
                default:
                printf("error: invalid vacancy\n");
                    break;
            }

            // update the total quantity of X, Y, Z
            X_total += report[m].allocation[id].schedule[0][2];
            Y_total += report[m].allocation[id].schedule[1][2];
```

```c
                Z_total += report[m].allocation[id].schedule[2][2];

            }

            // report the total quantity of X, Y, Z
            report[m].X[0] = currentX + 1;
            report[m].X[1] = X_total;
            report[m].Y[0] = currentY + 1;
            report[m].Y[1] = Y_total;
            report[m].Z[0] = currentZ + 1;
            report[m].Z[1] = Z_total;

        }

        if (strcmp(algorithmName, "FCFS") == 0) {
            // write the schedule to parent
            if (write(scheduler.fdc2p[1], &schedule[0], sizeof(struct Schedule)) < 0)
            {
                perror("error when writing schedule to parent");
                exit(EXIT_FAILURE);
            }
            // write the report to parent
            if (write(scheduler.fdc2p[1], &report[0], sizeof(struct Report)) < 0)
            {
                perror("error when writing report to parent");
                exit(EXIT_FAILURE);
            }
        } else if (strcmp(algorithmName, "SJF") == 0) {
            // write the schedule to parent
            if (write(scheduler.fdc2p[1], &schedule[1], sizeof(struct Schedule)) < 0)
            {
                perror("error when writing schedule to parent");
                exit(EXIT_FAILURE);
            }
            // write the report to parent
            if (write(scheduler.fdc2p[1], &report[1], sizeof(struct Report)) < 0)
            {
                perror("error when writing report to parent");
                exit(EXIT_FAILURE);
            }
        } else {
            int w = 0;
            float s = 0;
            for (int m = 0; m < 3; m++) {
                if(report[m].X[1] + report[m].Y[1] + report[m].Z[1] >= s) {
                    w = m;
                    s = report[m].X[1] + report[m].Y[1] + report[m].Z[1];
                }
            }
            // write the schedule to parent
            if (write(scheduler.fdc2p[1], &schedule[w], sizeof(struct Schedule)) < 0)
            {
                perror("error when writing schedule to parent");
                exit(EXIT_FAILURE);
            }
            // write the report to parent
            if (write(scheduler.fdc2p[1], &report[w], sizeof(struct Report)) < 0)
            {
                perror("error when writing report to parent");
                exit(EXIT_FAILURE);
            }
        }
        // close the pipe
        close(scheduler.fdp2c[0]);
        close(scheduler.fdc2p[1]);
        exit(0);
    }
    else
    {
        // input module now
        close(scheduler.fdp2c[0]);
        close(scheduler.fdc2p[1]);
        // write algorithm name to scheduler
        int messageLength = strlen(algorithm) + 1; // null terminator is also passed
        if (write(scheduler.fdp2c[1], &messageLength, sizeof(messageLength)) < 0)
        {
            perror("error when writing messageLength to scheduler");
            exit(EXIT_FAILURE);
        }
        if (write(scheduler.fdp2c[1], algorithm, messageLength) != messageLength)
        {
            perror("error when writing algorithm name to scheduler");
            exit(EXIT_FAILURE);
        }
        // write period to scheduler
        struct Period period;
        period.startDate = startDate;
        period.endDate = endDate;
        write(scheduler.fdp2c[1], &period, sizeof(period));
        // write orders to scheduler
        // number of orders to come
        if (write(scheduler.fdp2c[1], &orderNum, sizeof(orderNum)) < 0)
        {
            perror("error when writing orderNum to scheduler");
            exit(EXIT_FAILURE);
        }
        // orders
        if (write(scheduler.fdp2c[1], order, sizeof(struct Order) * orderNum) < 0)
        {
            perror("error when writing orders to scheduler");
            exit(EXIT_FAILURE);
        }
        // read schedule from scheduler
        struct Schedule schedule;
        // initialize schedule
        schedule.schedule_id = 0;
        for (int i = 0; i < MAX_ORDER_NUM; i++)
        {
            schedule.schedule_X[0][i] = -1;
            schedule.schedule_X[1][i] = -1;
            schedule.schedule_Y[0][i] = -1;
            schedule.schedule_Y[1][i] = -1;
            schedule.schedule_Z[0][i] = -1;
            schedule.schedule_Z[1][i] = -1;
        }
        if (read(scheduler.fdc2p[0], &schedule, sizeof(struct Schedule)) < 0)
        {
            perror("error when reading schedule from scheduler");
            exit(EXIT_FAILURE);
```

```c
        }
        // read report from scheduler
        struct Report report;
        // initlaize report
        for (int i = 0; i < MAX_ORDER_NUM; i++)
        {
            report.allocation[i].order_id = -1;
            report.allocation[i].accepted = -1;
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    report.allocation[i].schedule[j
    ][k] = -1;
                }
            }
        }
        if (read(scheduler.fdc2p[0], &report, sizeof
    (struct Report)) < 0)
        {
            perror("error when reading report from
    scheduler");
            exit(EXIT_FAILURE);
        }

        //close the pipe
        close(scheduler.fdp2c[1]);
        close(scheduler.fdc2p[0]);
        // wait for the scheduler process to
    terminate, then proceed
        int status;

        waitpid(scheduler.pid, &status, 0);
        // print the schedule to console
        printSchedule(schedule);
        // write the report to file
        writeReport(&report);
    }
}
int calcAccepted(struct Report *report)
{
    // calculate the number of accepted orders
    int accepted = 0;
    for (int i = 0; i < orderNum; i++)
    {
        if (report->allocation[i].accepted == 1)
        {
            accepted++;
        }
    }
    return accepted;
}
void writeReport(struct Report *report)
{
    int totalDays = diffDate(startDate, endDate) +
    1;
    FILE *file = fopen(reportFileName, "w");
    if (file == NULL)
    {
        perror("Error opening file");
        return;
    }
    fprintf(file, "***PLS Schedule Analysis Report
    ***\n\n");
    fprintf(file, "Algorithm used: %s\n\n",
    algorithmName);
    int accepted = calcAccepted(report);
    int rejected = orderNum - accepted;
    // first print accepted orders
    fprintf(file, "There are %d Orders ACCEPTED.
    Details are as follows:\n\n", accepted);
    // print column names
    // first column is order number, width 16
    // second column is start date, width 14
    // third column is end date, width 14
    // fourth column is days, width 8
    // fifth column is quantity, width 12
    // sixth column is plant, width 9
    fprintf(file, "%16s%14s%14s%8s%12s%9s\n", "ORDER
 NUMBER", "START", "END", "DAYS", "QUANTITY", "
PLANT");
    fprintf(file, "
==========================================\n");
    for (int i = 0; i < orderNum; i++)
    {
        if (report->allocation[i].accepted == 0)
        {
            continue;
        }
        if (report->allocation[i].accepted == -1)
        {
            perror("Error: allocation not done.");
            exit(1);
        }

        // first print its schedule at plant x
        if (report->allocation[i].schedule[0][1] >
0){
            struct tm orderStartDate = calcDate(
startDate, report->allocation[i].schedule[0][0])
;
            char orderStartDateStr[MAX_DATE_LEN];
            dateToStr(orderStartDate,
orderStartDateStr);
            struct tm orderEndDate = calcDate(
orderStartDate, report->allocation[i].schedule
[0][1] - 1);
            char orderEndDateStr[MAX_DATE_LEN];
            dateToStr(orderEndDate, orderEndDateStr)
;
            fprintf(file, "%16s%14s%14s%8d%12d%9s\n"
, order[i].order_number, orderStartDateStr,
orderEndDateStr, report->allocation[i].schedule
[0][1], report->allocation[i].schedule[0][2], "
Plant_X");
        }
        // then print its schedule at plant y
        if (report->allocation[i].schedule[1][1] >
0){
            struct tm orderStartDate = calcDate(
startDate, report->allocation[i].schedule[1][0])
;
            char orderStartDateStr[MAX_DATE_LEN];
            dateToStr(orderStartDate,
orderStartDateStr);
            struct tm orderEndDate = calcDate(
orderStartDate, report->allocation[i].schedule
[1][1] - 1);
            char orderEndDateStr[MAX_DATE_LEN];
            dateToStr(orderEndDate, orderEndDateStr)
;
            fprintf(file, "%16s%14s%14s%8d%12d%9s\n"
, order[i].order_number, orderStartDateStr,
orderEndDateStr, report->allocation[i].schedule
[1][1], report->allocation[i].schedule[1][2], "
Plant_Y");
        }
        // finally print its schedule at plant z
        if (report->allocation[i].schedule[2][1] >
0){
            struct tm orderStartDate = calcDate(
startDate, report->allocation[i].schedule[2][0])
;
            char orderStartDateStr[MAX_DATE_LEN];
            dateToStr(orderStartDate,
orderStartDateStr);
            struct tm orderEndDate = calcDate(
orderStartDate, report->allocation[i].schedule
```

```c
[2][1] - 1);
            char orderEndDateStr[MAX_DATE_LEN];
            dateToStr(orderEndDate, orderEndDateStr);
            fprintf(file, "%16s%14s%14s%8d%12d%9s\n", order[i].order_number, orderStartDateStr, orderEndDateStr, report->allocation[i].schedule[2][1], report->allocation[i].schedule[2][2], "Plant_Z");
        }
    }
    fprintf(file, "- End -\n\n");
    fprintf(file, "================================\n");
    fprintf(file, "\n\n");
    // then print rejected orders
    fprintf(file, "There are %d Orders REJECTED. Details are as follows:\n\n", rejected);
    // print column names
    // first column is order number, width 16
    // second column is product name, width 11
    // third column is due date, width 14
    // fourth column is quantity, width 12
    fprintf(file, "%16s%11s%14s%12s\n", "ORDER NUMBER", "PRODUCT", "DUE DATE", "QUANTITY");
    fprintf(file, "=======================================\n");
    for (int i = 0; i < orderNum; i++)
    {
        if (report->allocation[i].accepted == 1)
        {
            continue;
        }
        if (report->allocation[i].accepted == -1)
        {
            perror("Error: allocation not done.");
            exit(1);
        }
        char orderProductName[10] = "Product_";
        char temp[2] = {order[i].product_name, '\0'};
        strcat(orderProductName, temp);
        char orderDueDateStr[MAX_DATE_LEN];
        dateToStr(order[i].dueDate, orderDueDateStr);
        fprintf(file, "%16s%11s%14s%12d\n", order[i].order_number, orderProductName, orderDueDateStr, order[i].order_quantity);
    }
    fprintf(file, "- End -\n\n");
    fprintf(file, "=======================================\n");
    fprintf(file, "\n\n");
    // now write the performance analysis
    fprintf(file, "***PERFORMANCE\n\n");
    // first print the performance of plant x
    fprintf(file, "Plant_X\n");
    // the bench marks are indented by 4 spaces
    // the data are right aligned with width 10
    fprintf(file, "       %-35s%10d days\n", "Number of days in use:", report->X[0]);
    fprintf(file, "       %-35s%10d (in total)\n", "Number of products produced:", report->X[1]);
    fprintf(file, "       %-35s%10.2f %%\n", "Utilization of the plant:", (double)report->X[1] / (totalDays * 300.0) * 100);
    fprintf(file, "\n");
    // then print the performance of plant y
    fprintf(file, "Plant_Y\n");
    fprintf(file, "       %-35s%10d days\n", "Number of days in use:", report->Y[0]);
    fprintf(file, "       %-35s%10d (in total)\n", "Number of products produced:", report->Y[1]);
    fprintf(file, "       %-35s%10.2f %%\n", "Utilization of the plant:", (double)report->Y[1] / (totalDays * 400.0) * 100);
    fprintf(file, "\n");
    // finally print the performance of plant z
    fprintf(file, "Plant_Z\n");
    fprintf(file, "       %-35s%10d days\n", "Number of days in use:", report->Z[0]);
    fprintf(file, "       %-35s%10d (in total)\n", "Number of products produced:", report->Z[1]);
    fprintf(file, "       %-35s%10.2f %%\n", "Utilization of the plant:", (double)report->Z[1] / (totalDays * 500.0) * 100);
    fprintf(file, "\n");
    // overall performance
    fprintf(file, "%-42s%10.2f %%\n", "Overall of utilization:", (double)(report->X[1] + report->Y[1] + report->Z[1]) / (totalDays * 1200.0) * 100);
    // fprintf(file, "Overall of utilization: %.2f %%\n", (double)(report->X[1] + report->Y[1] + report->Z[1]) / (totalDays * 1200) * 100);
    fclose(file);
}


void init(){
    //clear the content of the file for invalid inputs
    FILE *filePtr;
    // Open the file in append mode
    filePtr = fopen(INVALID_INPUTS, "w");
    if (filePtr == NULL) {
        perror("Error opening file");
        return;
    }
    // Close the file
    fclose(filePtr);
}
int main()
{
    init();
    inputModule();
    return 0;
}
```

Listing 3. PLS Source Code

B. Sample Outputs

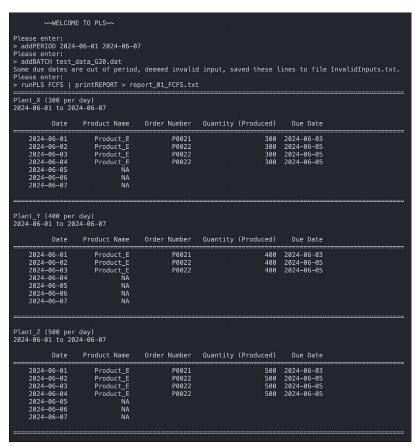


Fig. 5. This is the output on the terminal of FCFS algorithm.