**Semester Project Report**

**Study Scheduling System (S3)**

Comp 2432 Operating System

The Hong Kong Polytechnic University

Department of Computing

Group 7

XU Xiaochuan 17082834D

Han Shanglin 17083504D

He Jiashu 17082689D

Lv Zhian 17083718D

Lian Xiang 17082482D

10 April 2019

COMP2432 Operating System – Project Report

1. ***Introduction***

Nowadays, people’s life is much busier than ever before. Thus, the efficiency becomes an important topic in our daily life. As a student, we have to deal with a huge number of tasks as the semester goes on. In order to arrange all these tasks to achieve the best result in an efficient manner, our program aims at automatically scheduling the tasks for the user to help them make best use of their time and achieve the best result. Meanwhile, through reflecting from scheduling the tasks, we also aim at obtaining better and deeper understanding of schedule mechanism in operating system.

1. ***Scope (What operating systems topics have been covered in this project?)*** 
   * Since the daily life task scheduling is similar to CPU scheduling, some CPU scheduling algorithms covered in the lectures including FCFS, SJF, SRT, EDF are used to realize the task arrangement.
   * Multiple processes are used to make the program more efficient.
   * Pipe is used to transfer all information about the tasks between processes
2. ***Concept (What are the algorithms behind?)***

*FCFS*

First Come First Serve is the simplest and easiest scheduling algorithm. In this algorithm, the CPU is allocated to the processes in the order they request it. The implementation of FCFS is easily done with a queue (a FIFO structure). When the first process enters the system, it starts its execution immediately and runs till it completes its execution or it meets its deadline and still not finishes. As other processes enter the system, they are put at the end of the queue and wait to get the CPU. When a process finishes executing, it releases the CPU and is removed from the queue and the CPU is allocated to next process at the head of the queue.

*Priority*

Priority scheduling puts the tasks in a sorted linked list according to their priority (Project > Assignment > Revision > Other activites), then allocates time slots to the tasks according to their position in the linked list, from head to tail. For any 2 tasks share the same priority, the earlier one will be executed first.

*Deadline*

Earliest deadline first is dynamic priority scheduling algorithm for real time embedded systems where priority is the deadline of each task. In this algorithm, assignments and projects will be executed by comparing their deadlines, the earlier the deadline, the earlier a task will be allocated with time slots. The revision and other activities will be put backward.

One thing to mention is it uses the same set of implementation as the priority, only changes how to compare the priority of 2 tasks. In the program, the function receives a flag parameter, 0 to denote priority, 1 to denote deadline as priority.

***Your own scheduling algorithm (if any)***

Our scheduling algorithm named "smarter DDL fighter". This algorithm is base on the concept that "project" and "assignment" are much important than the "revision" and "activity". Also, sometimes it still has time in the end, but no "revision" and "activity" can do. Due to the previous case, we design a much "smarter" DDL fighter.

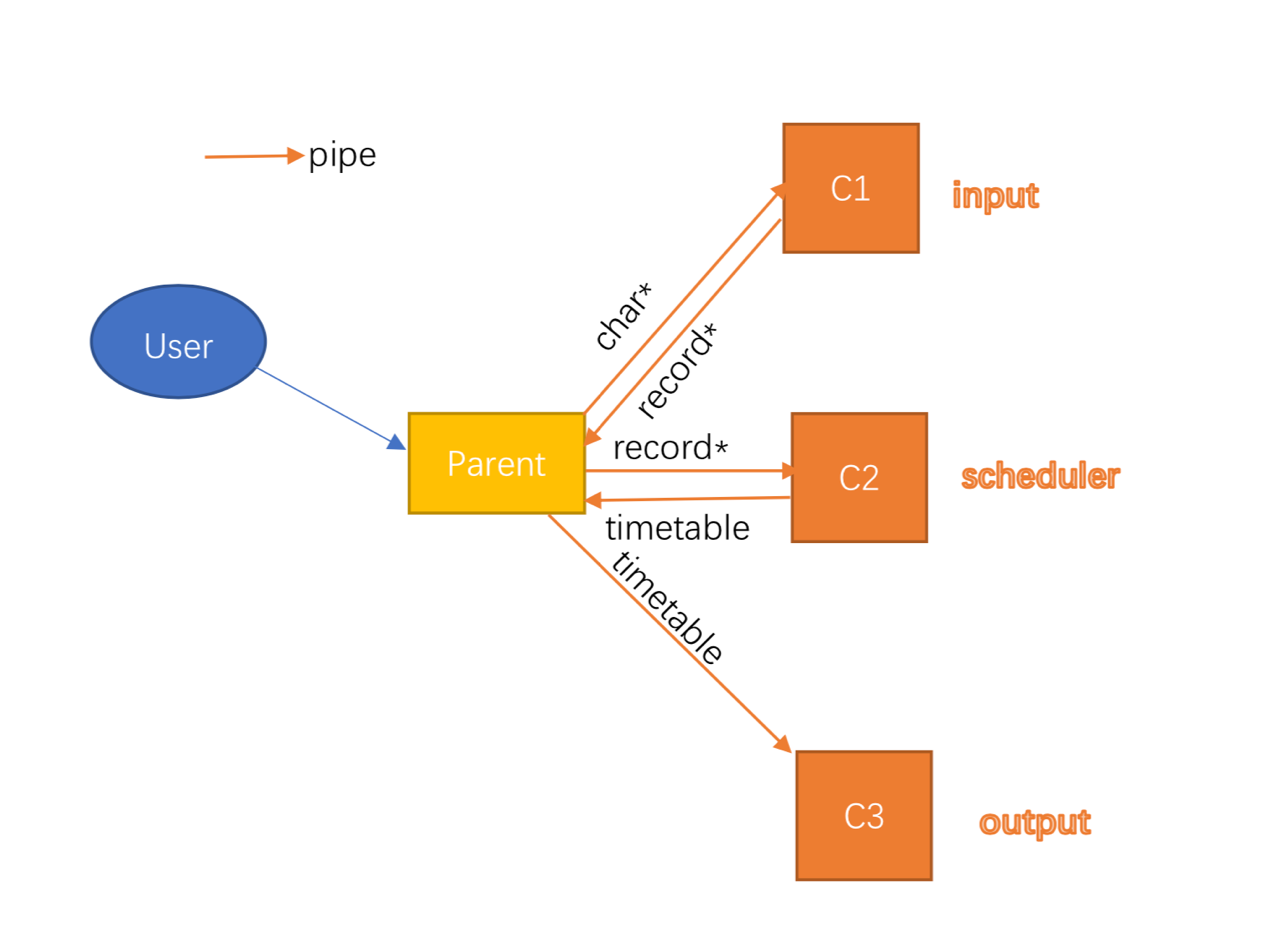
This algorithm first sorts all events by the deadline or start time, then try to arrange the time for projects and assignments and generate a time layer which is used for deciding how much free time remain in each time range. The critical point is that in this step, range the time range for projects and assignments, but not a specific time. For example, there is a project with 2-hour cost and due on the first day's end. In this step, arrange that do the project on the first day.

After the time layer generated, the algorithm begins to arrange revisions and activities. For each event of these two types, checking is there enough free time to organize. If so, kindly do this event and update the time layer. Otherwise, reject the event.

Finally, fill the project and assignment in the arrange table. For each accepted event, seek a free time slot for it.

1. ***Software structure of your system***

*Original Version*

**

In our original vision, the parent process will fork three children (called C1 C2 C3) for processing user input, doing schedule algorithm and printing timetable respectively. The design idea is from the MVC pattern.

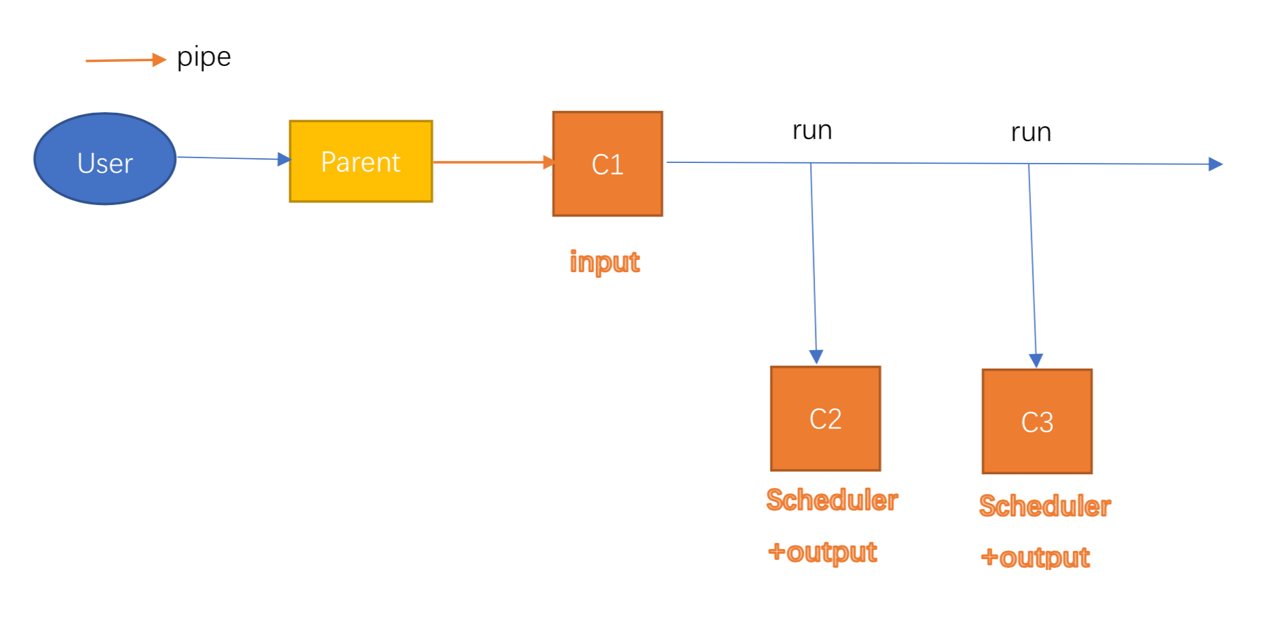
Each time the parent receives user input, it will pass the input message to C1 through a pipe. Then C1 will parser these user input message and return the record result in a specific format to parent through another pipe.

After the parent received the record from C1, it will pass forward to C2. Then C2 will store the content and run schedule algorithm to arrange the timetable once user needs. Then C2 return the time table to the parent through a pipe.

Once the user asks for the timetable, the parent receives from user input and passes to C1. Then got the result from C1 and sent it to C2. The C2 will run the algorithm and give the result back to the parent. The parent will pass the time table to another child C3. Then C3 will do the printing work.

However, this kind of design style has a nonnegligible drawback. Since we can only pass characters through the pipe, all other data type or data structure should be transformed into character form to be passed. Then the receiver has to convert it back to the original data type or data structure, which spend a lot of CPU time on unnecessary format conversions. In this case, we consider that the parent itself can do c1's job. And the printing work can be done by C2 directly to avoid passing the time table data structure twice through the pipe. So, C1 and C3 are not needed anymore. Thus, we get our second version as follows.

*Final Version:*



Following our analysis of the original version, first, we think about the scheduling algorithm part, which is the critical core of the project. The algorithm may be not isolation, so once the algorithm runs, it may pollute the original data. And if the program reruns the algorithm, it may cause a different result. So we decided to use a child process to handle the algorithm.

Second, we found that if the time table passes between scheduling algorithm process, the parent process and the logging process, many CUP time spent on the useless format converting. Therefore, we decided to delete the logging process and do the logging method in the scheduling process after the timetable is generated.

Finally, we found that the bottleneck of the program is user input. Imagine a scenario where the user enters an "addBatch" instruction that the program needs to process for a long time. The user must wait for the program to complete before proceeding to the next step. We think this kind of situation is not good enough. So we think about asynchronous I/O. We decide to use an additional process to deal with this problem. Finally, our design is as follows.

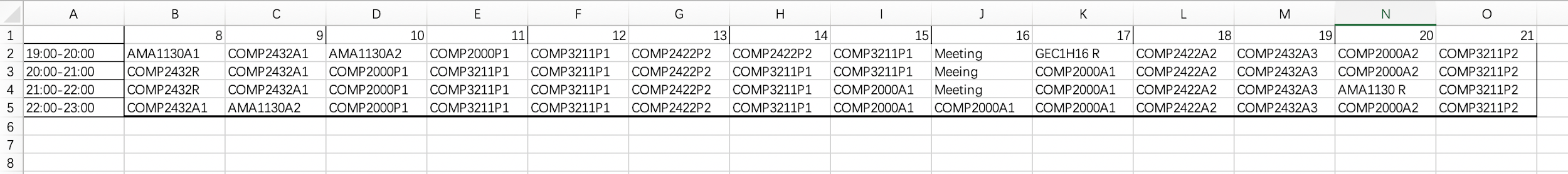
The parent will fork a child process(C1) to process user input. Once the parent process receives an instruction, it will pass to C1. And the parent only does the wait-pass job. For C1, it will continue receiving content from the parent process, and process it. Once the instruction is "Run algorithm" C1 will fork a new child process (C2) and let C2 do the scheduling and logging job. After C2 finish scheduling and logging, it will exit. This procedure prevents algorithm contamination data and avoids excess data transfer.

1. ***Testing cases (What have you done to test the correctness of the program/application?)***

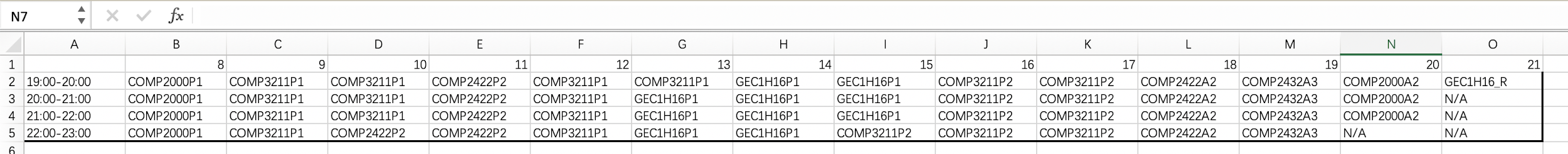
We write test cases and draw the correct schedule according to the algorithm, then compare the result of our program with the correct schedule drawn by ourselves.

You can also check the excel file in our project packet.

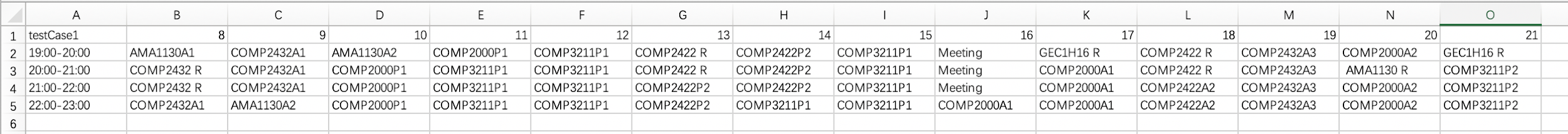
**FCFS:**



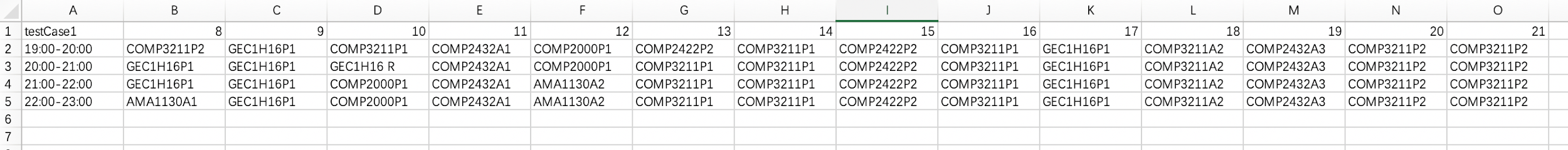
**Priority:**

****

**Deadline:**

****

**SDDL(our own algorithm):**

****

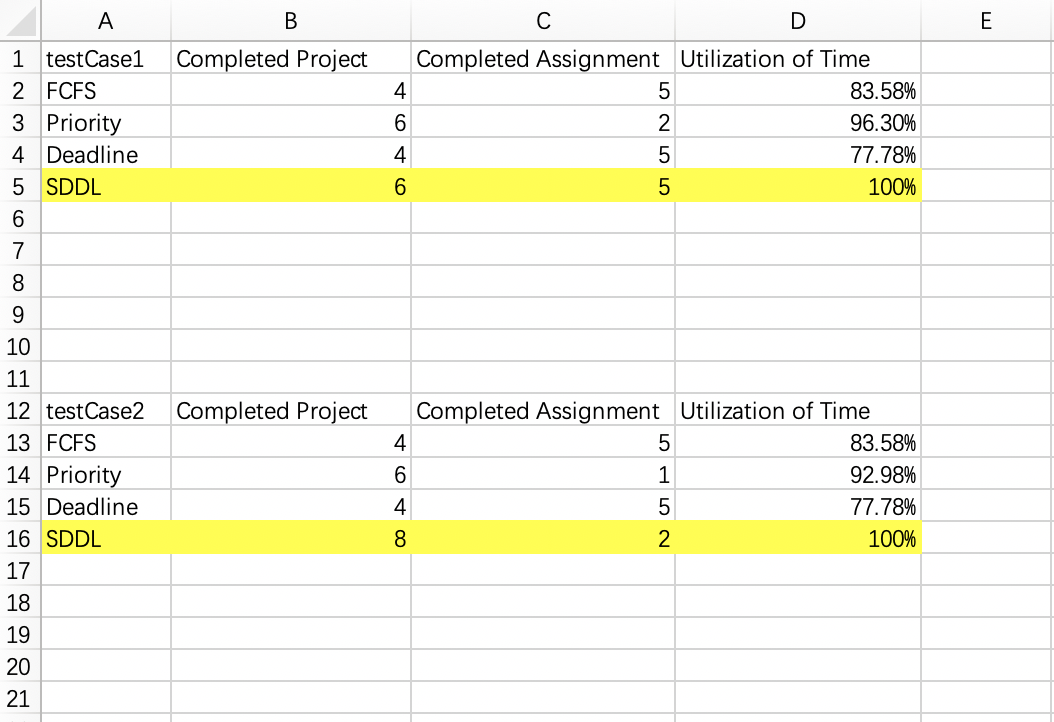
1. ***Performance analysis (Discuss and analyze the algorithm used in the program. Why it is better than the others?)***

In order to compare the performance of all algorithms in our program, we consider three aspects as following:

1. the number of completed projects
2. the number of assignments as following
3. the utilization of time

The one with more completed projects is considered better. If the number of completed projects are the same, the one with more completed assignments is better. If the number of completed assignments are still the same, then the one with higher utilization rate of time is better.

In the analysis, we found out that our smart deadline algorithm always performs much better than all other algorithms according to our evaluation criteria.



As you can see from this table, the SDDL always complete the most completed projects. We also found out that the smart deadline algorithm has an extremely high time utilization rate.

1. ***Program set up and execution (How to compile and execute your project? On which Linux server would you like your project to be executed?)***

1) Use apollo to test our program

2) download our work

3) unzip the file

4) Compile the c files

*command: gcc std=gnu99 -o projectS3 S3.c*

5) Run the program

*command: ./ProjectS3*

6) To test our program, you can use our test case

*Please enter: addBatch S3\_data\_1.dat*

7) Choose an algorithm to run the test case (This example use FCFS, output to a text file called out1.txt)

*Please enter: runS3 FCFS FCFS\_data1\_out*

8) The schedule and summary report are generated

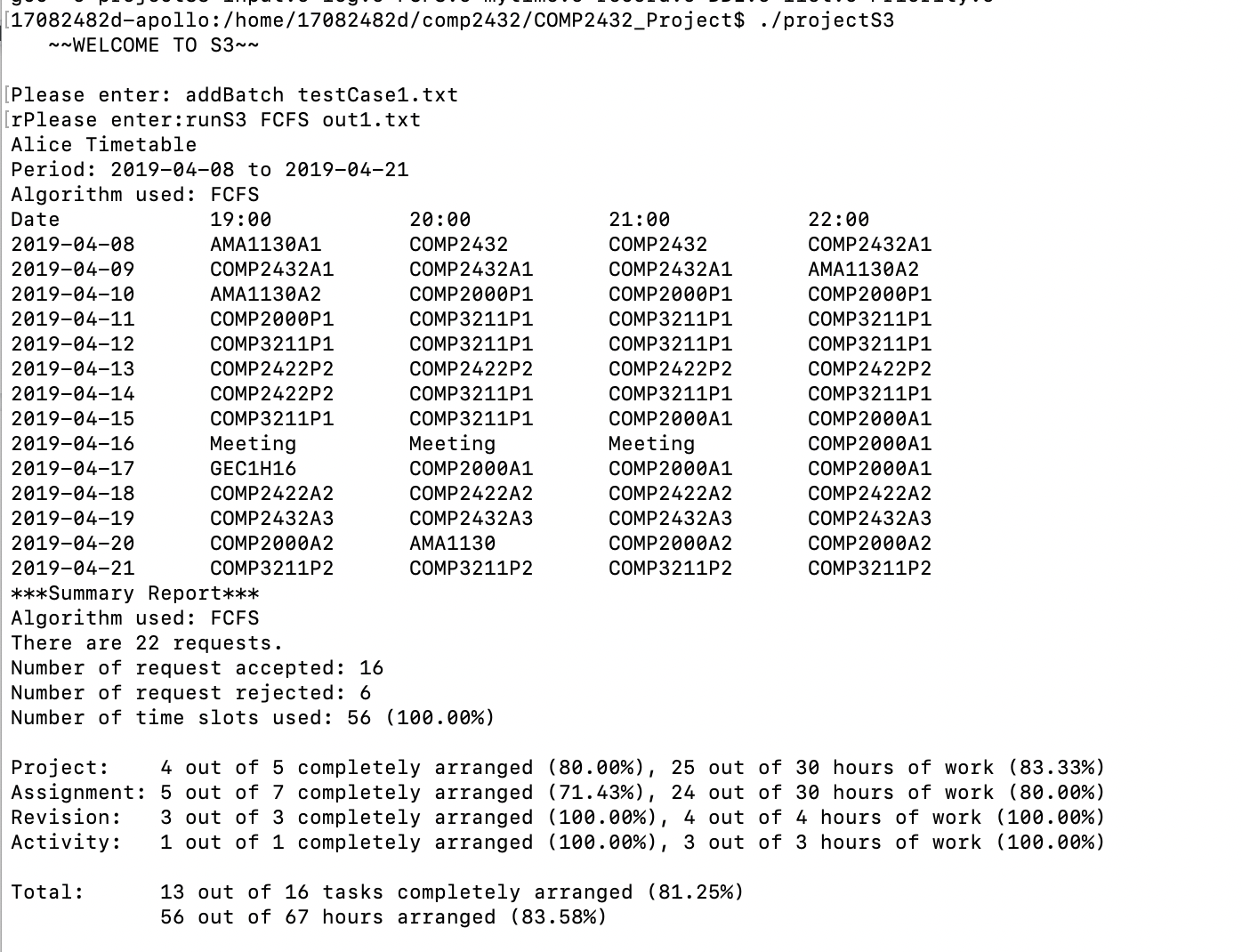
9) If you want to exit

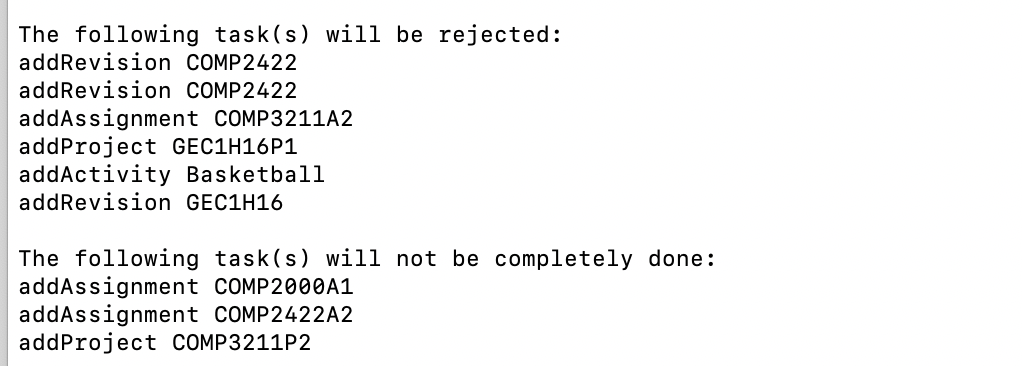
*Please enter: exitS3*

1. ***Appendix I – source code, and, soft copies of “Time Table”, “Summary Report” and “Log File”.***

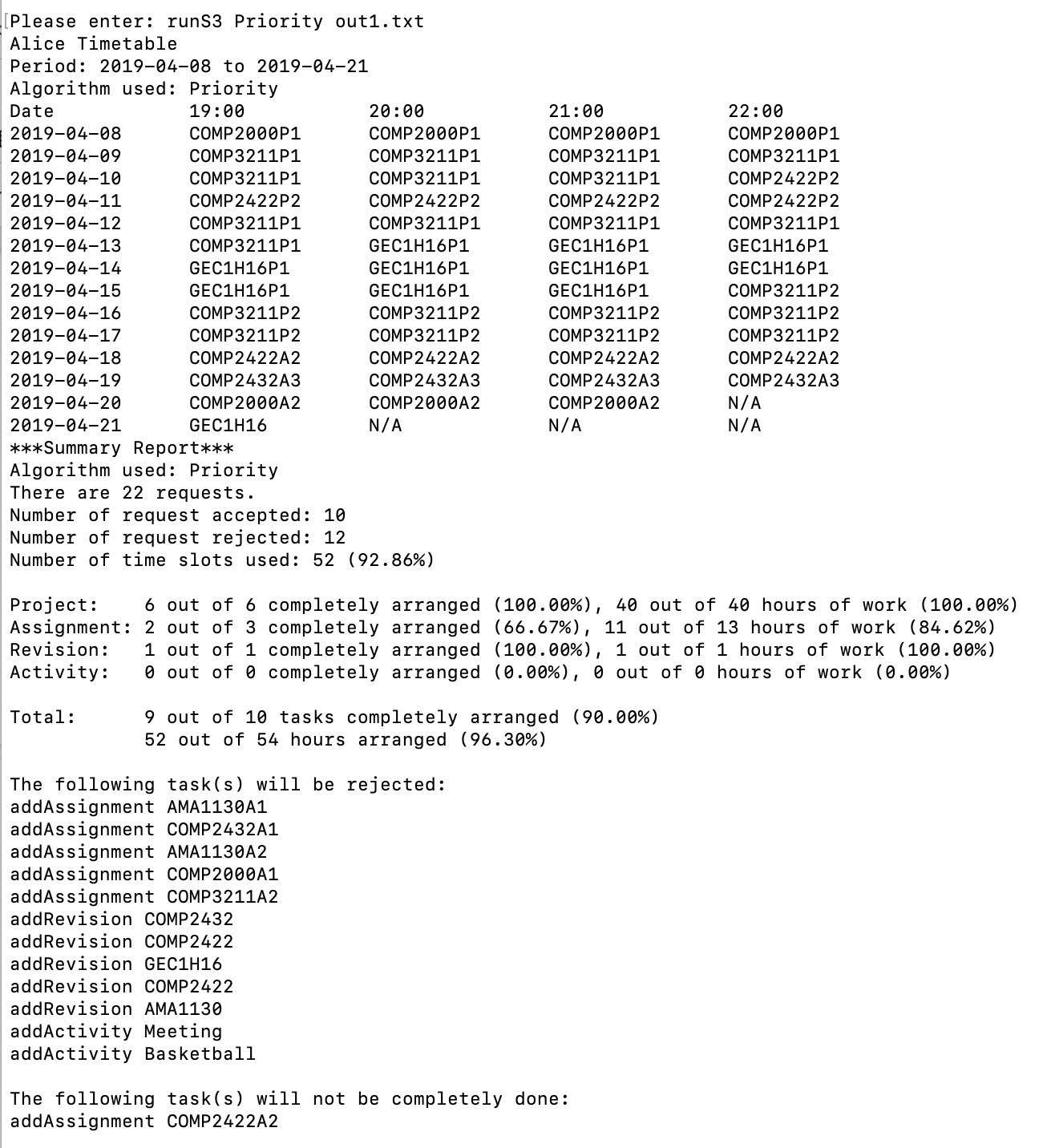
Source code: please refer to our project packet

***FCFS:***

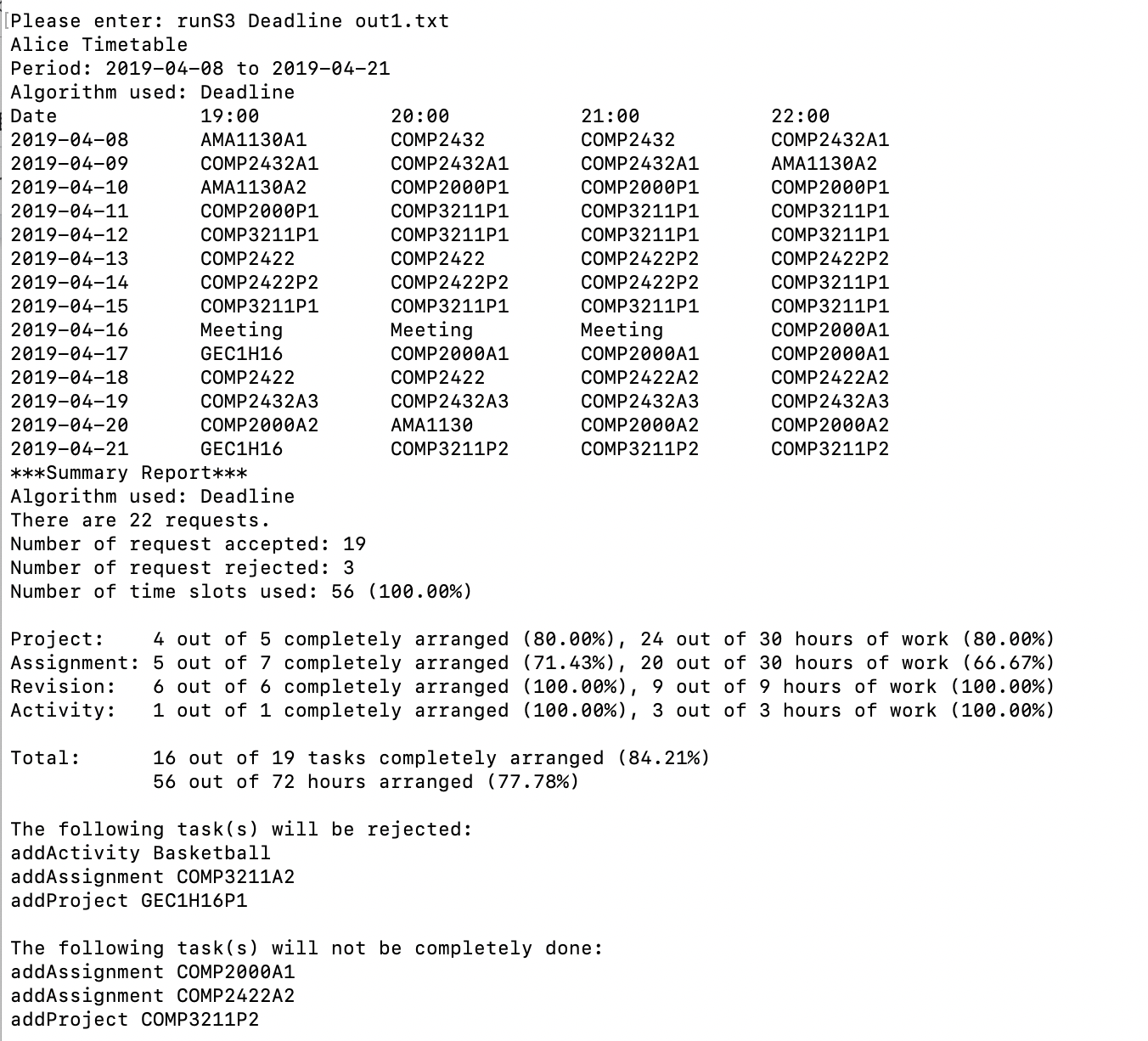
******

******

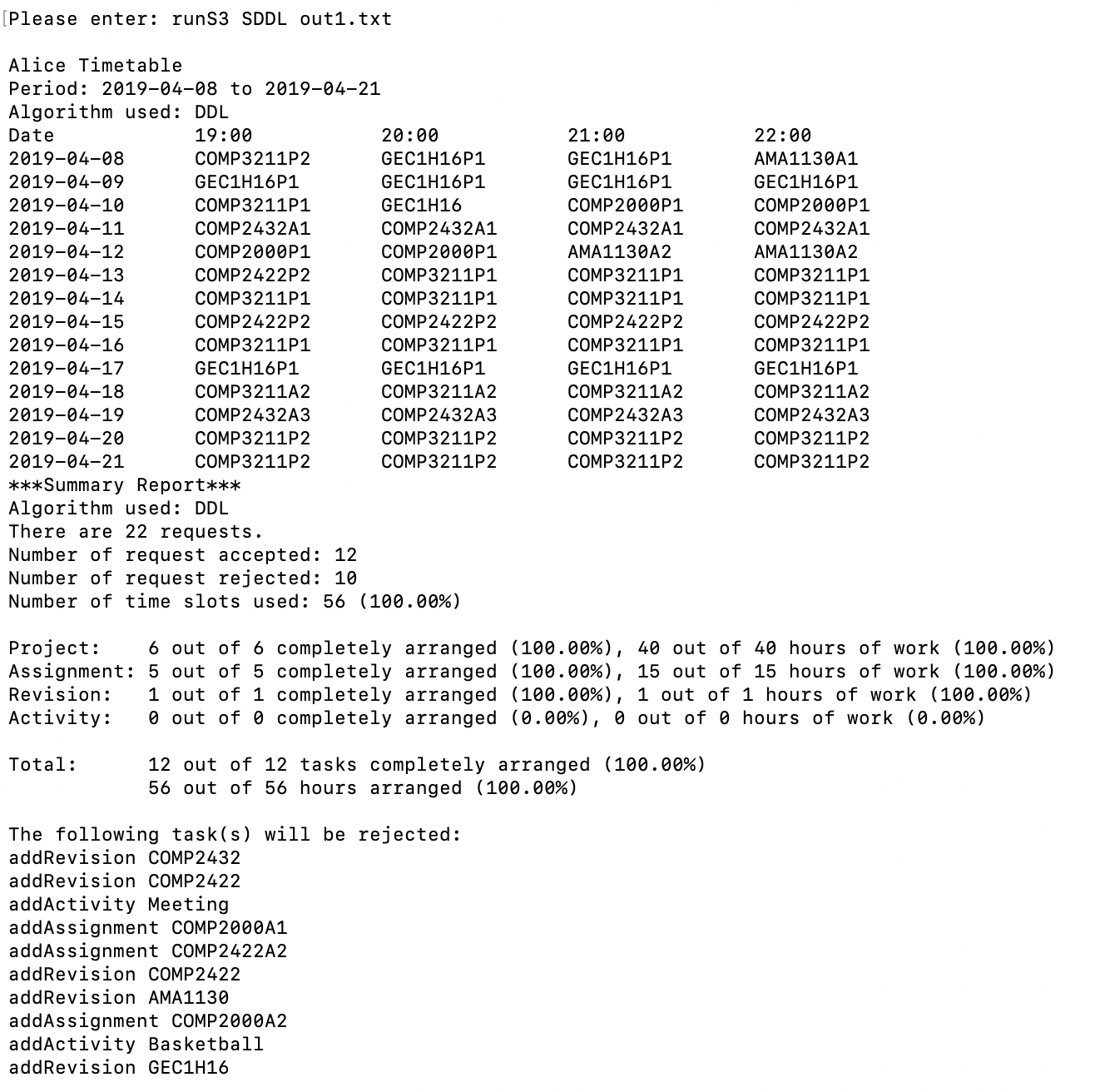
***Priority:***

******

***Deadline:***

******

***SDDL:***

******

1. ***Appendix II – Contribution of Work.***

|  |  |  |
| --- | --- | --- |
| **Name** | **Student ID** | **Work** |
| Han Shanglin | 17083504d | Algorithm & Data Structure & Documentation |
| He Jiashu | 17082689d | Algorithm & Documentation |
| Lian Xiang | 17082482d | Test Case Generation and Answer & Documentation |
| Lyu Zhian | 17083718d | Print & Log File |
| Xu Xiaochun | 17082834d | Input & pipe/fork handling & Documentation |