Project 5: CPU Scheduling Simulator

Ethan Cook

Fall 2023, CMPE

1 Introduction

Students were asked to implement a simple CPU scheduler, which operates three CPU scheduling policies, First Come First Serve (FCFS), Round Robin (RR), and Shortest Remaining Time First (SRTF). The program should take a task list file and command-line arguments as input. Using the specified inputs, the program should output time-slots and their running task. Then, the program computes statistics of the policy being used and outputs it. Students were suggested to design the system using a data flow diagram (DFD), then implement it in C programming. Note that the words task and process are used interchangeably throughout this report.

2 System Design

The first step I took to implementing the CPU scheduler is laying out my initial design. I based it off of Dr. Xiao Qin's sample data flow diagram. My DFD is shown in Figure 1. Note that functions and data are in black while the C files are in blue.

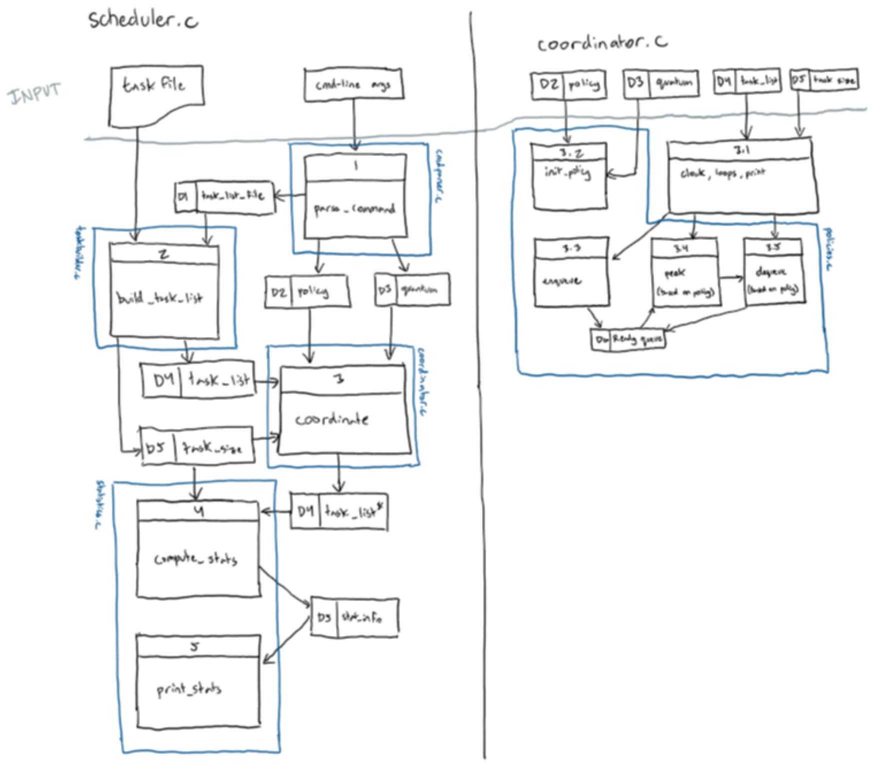
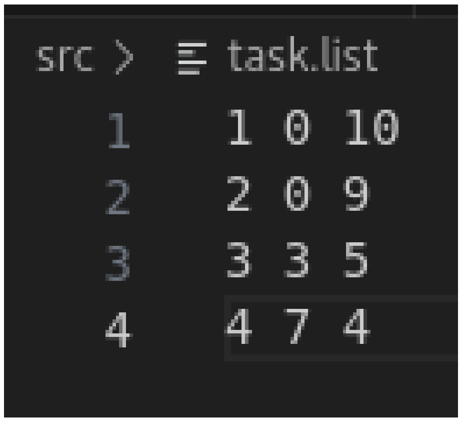


Figure 1: Data Flow Diagram

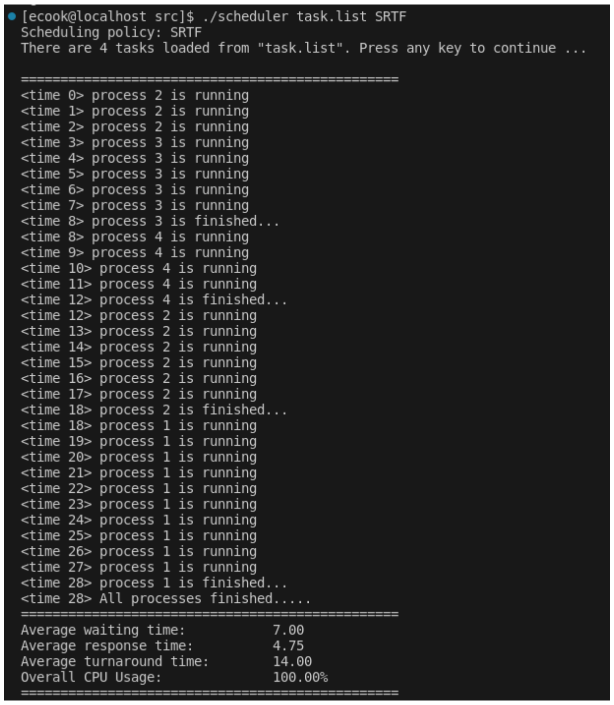
3 Solution Description

The CPU scheduling simulator takes a task list file as input. The task list file should contain lines made up of three integers separated by spaces where each line is a task. The first integer represents the process id, the second: arrival time, and the third: CPU time. A sample task list file would look like Figure 2a. In order to run the scheduling simulator, one must enter the following command: "./scheduler (task_list_file) [FCFS/RR/SRTF] [quantum]". Using the sample task list file from Figure 2a and the policy SRTF, the scheduler would print the results in Figure 2b.



```
1 0 10
2 0 9
3 3 5
4 7 4
```

(a) Task list



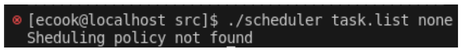
```
[ecook@localhost src]$ ./scheduler task.list SRTF
Scheduling policy: SRTF
There are 4 tasks loaded from "task.list". Press any key to continue ...

================================================
<time 0> process 2 is running
<time 1> process 2 is running
<time 2> process 2 is running
<time 3> process 3 is running
<time 4> process 3 is running
<time 5> process 3 is running
<time 6> process 3 is running
<time 7> process 3 is running
<time 8> process 3 is finished...
<time 8> process 4 is running
<time 9> process 4 is running
<time 10> process 4 is running
<time 11> process 4 is running
<time 12> process 4 is finished...
<time 12> process 2 is running
<time 13> process 2 is running
<time 14> process 2 is running
<time 15> process 2 is running
<time 16> process 2 is running
<time 17> process 2 is running
<time 18> process 2 is finished...
<time 18> process 1 is running
<time 19> process 1 is running
<time 20> process 1 is running
<time 21> process 1 is running
<time 22> process 1 is running
<time 23> process 1 is running
<time 24> process 1 is running
<time 25> process 1 is running
<time 26> process 1 is running
<time 27> process 1 is running
<time 28> process 1 is finished...
<time 28> All processes finished.....
================================================
Average waiting time:           7.00
Average response time:          4.75
Average turnaround time:        14.00
Overall CPU Usage:              100.00%
================================================
```

(b) Output

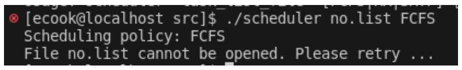
Figure 2: Sample simulation

If a user were to enter arguments that could potential cause errors, my program will let the user know where an error is in the argument. I implemented these argument and error handlers in the command-line parser. Two examples are shown in Figure 3. Once a command has been entered with valid arguments and a valid task list file, the CPU scheduling simulator can run based on the policy entered.



```
[ecook@localhost src]$ ./scheduler task.list none
Sheduling policy not found
```

(a) Policy input error



```
[ecook@localhost src]$ ./scheduler no.list FCFS
Scheduling policy: FCFS
File no.list cannot be opened. Please retry ...
```

(b) Task list file error

Figure 3: Command-line errors

In order to do implement scheduling policies, I had to separate scheduling mechanisms from scheduling policies. As shown in Figure 1, the coordinator takes care of many of the scheduling mechanisms. The coordinate function calls when to add tasks to the ready queue, when to peek at the first task according to the policy, and when to delete a task from the ready queue. The peek function takes the ready queue and finds the best task to run based on the policy. I was able to implement the three scheduling policies by creating their own peek functions. For example, the SRTF peek function would look through all the tasks in the ready queue and return the task with the lowest remaining time. In order to understand which policies do better with different task lists, I needed to calculate the waiting times, response times, and turnaround times of each task. I was able to do this by logging the start time and finish time of each task, then in my compute_statistics function, I would calculate the times by using the following equations:

$$\begin{aligned} waiting\_time &= finish\_time - arrival\_time - cpu\_time \\ response\_time &= start\_time - arrival\_time \\ turnaround\_time &= finish\_time - arrival\_time \end{aligned}$$

Ideally, the user should see the averages of these values, so my program adds the waiting times, response times, and turnaround times for each task and divides by the task list size to get the averages.

4 Generality and Error Checking

My CPU scheduling simulator can be easily implemented in many different systems. The design is quite modular and easy to change. Adding new scheduling policies is as simple as updating the policy enumerator, adding a new peek policy, and updating the input error handling. My program will handle many input errors for the command line. Users will be prompted with text information if the command-line arguments are in any way incorrect. Although all command-line input errors are handled, I did not handle errors from task list files. If a user were to try to run the scheduler with a task list file that has characters in it or more than three integers, the program will crash. For example, if a task list file had one line that contained two integers instead of three, the scheduler will run without stopping due to a task not having a valid CPU time.

5 Miscellaneous Factors

I wrote my code to be easy to follow and readable. Variable names are concise yet descriptive. Each file contains block commenting in the heading and the code itself does not exceed 100 lines for each file. The comments include documentation for outside sources. Although I stayed away from innovative ideas outside the project given, I did create a header file for types. I defined types and constants used throughout the program and included the header file "types.h" in each file. This allowed me to implement unique types with ease.

6 Conclusion

This project deepened my knowledge in operating systems and expanded my skills in C programming along with sparking my interest in system design. I thoroughly enjoyed creating this CPU scheduling simulator and look forward to learning more about operating systems by implementation.