CPU Scheduling report

Sid Suresh

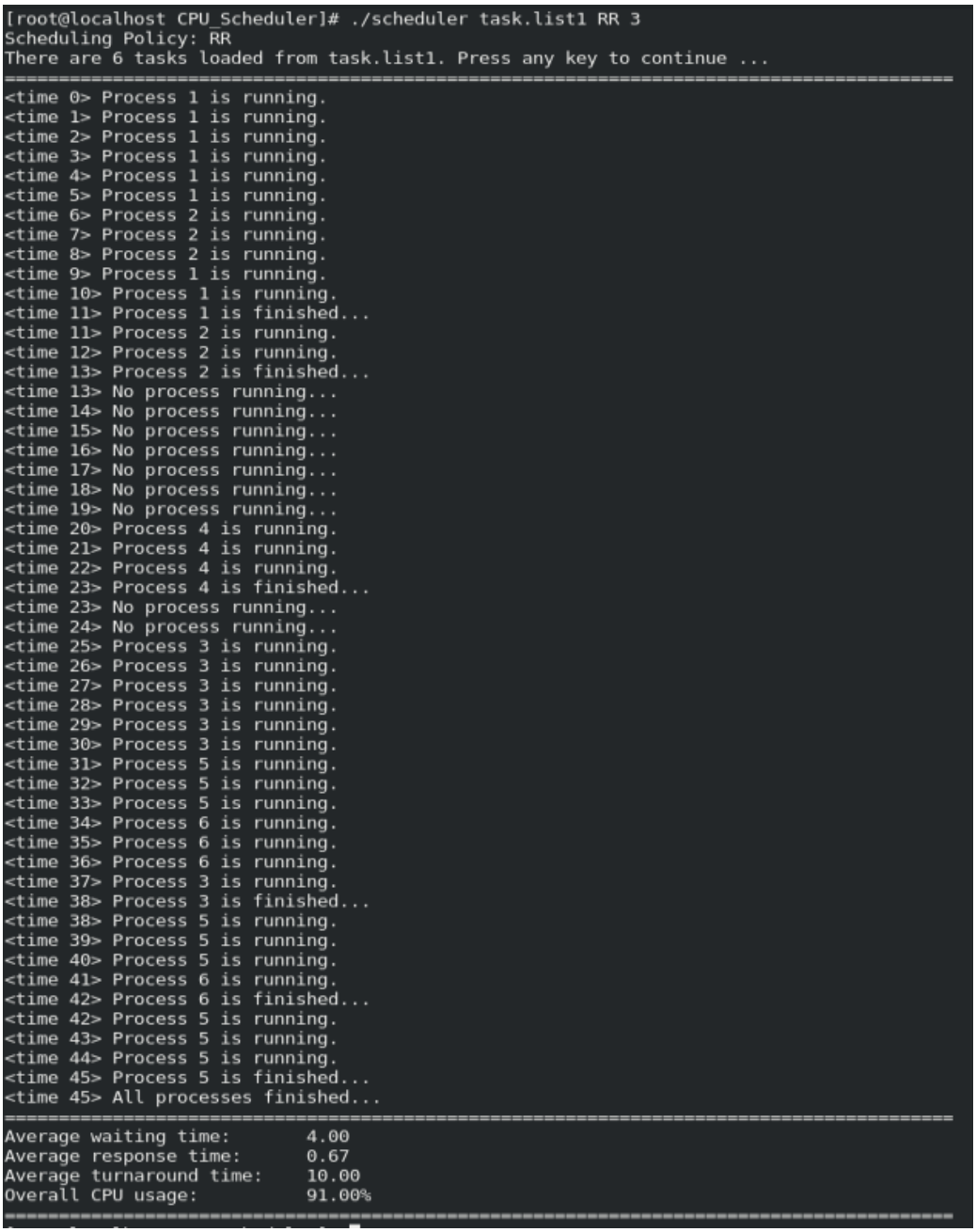
The operation of the program was tested using 2 input files: task.list and task.list1. task.list was used to simply test the function of the program while task.list1 was created to test edge cases like accounting for the time no process accessed the CPU. The input and output for each policy type are shown below.

```
[root@localhost CPU_Scheduler]# more task.list
1 0 10
2 0 9
3 3 5
4 7 4
5 10 6
6 10 7
[root@localhost CPU_Scheduler]#
```

```
[root@localhost CPU_Scheduler]# more task.list1
1 0 8
2 6 5
3 25 7
4 20 3
5 30 9
6 31 4
```

```
[root@localhost CPU_Scheduler]# ./scheduler task.list1 FCFS
Scheduling Policy: FCFS
There are 6 tasks loaded from task.list1. Press any key to continue ...
================================================================================
<time 0> Process 1 is running.
<time 1> Process 1 is running.
<time 2> Process 1 is running.
<time 3> Process 1 is running.
<time 4> Process 1 is running.
<time 5> Process 1 is running.
<time 6> Process 1 is running.
<time 7> Process 1 is running.
<time 8> Process 1 is finished...
<time 8> Process 2 is running.
<time 9> Process 2 is running.
<time 10> Process 2 is running.
<time 11> Process 2 is running.
<time 12> Process 2 is running.
<time 13> Process 2 is finished...
<time 13> No process running...
<time 14> No process running...
<time 15> No process running...
<time 16> No process running...
<time 17> No process running...
<time 18> No process running...
<time 19> No process running...
<time 20> Process 4 is running.
<time 21> Process 4 is running.
<time 22> Process 4 is running.
<time 23> Process 4 is finished...
<time 23> No process running...
<time 24> No process running...
<time 25> Process 3 is running.
<time 26> Process 3 is running.
<time 27> Process 3 is running.
<time 28> Process 3 is running.
<time 29> Process 3 is running.
<time 30> Process 3 is running.
<time 31> Process 3 is running.
<time 32> Process 3 is finished...
<time 32> Process 5 is running.
<time 33> Process 5 is running.
<time 34> Process 5 is running.
<time 35> Process 5 is running.
<time 36> Process 5 is running.
<time 37> Process 5 is running.
<time 38> Process 5 is running.
<time 39> Process 5 is running.
<time 40> Process 5 is running.
<time 41> Process 5 is finished...
<time 41> Process 6 is running.
<time 42> Process 6 is running.
<time 43> Process 6 is running.
<time 44> Process 6 is running.
<time 45> Process 6 is finished...
<time 45> All processes finished...
================================================================================
Average waiting time:       2.33
Average response time:      2.33
Average turnaround time:    8.33
Overall CPU usage:          91.00%
================================================================================
```

```
[root@localhost CPU_Scheduler]# ./scheduler task.list SRTF
Scheduling Policy: SRTF
There are 6 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 5 is running.
<time 13> Process 5 is running.
<time 14> Process 5 is running.
<time 15> Process 5 is running.
<time 16> Process 5 is running.
<time 17> Process 5 is running.
<time 18> Process 5 is finished...
<time 18> Process 2 is running.
<time 19> Process 2 is running.
<time 20> Process 2 is running.
<time 21> Process 2 is running.
<time 22> Process 2 is running.
<time 23> Process 2 is running.
<time 24> Process 2 is finished...
<time 24> Process 6 is running.
<time 25> Process 6 is running.
<time 26> Process 6 is running.
<time 27> Process 6 is running.
<time 28> Process 6 is running.
<time 29> Process 6 is running.
<time 30> Process 6 is running.
<time 31> Process 6 is finished...
<time 31> Process 1 is running.
<time 32> Process 1 is running.
<time 33> Process 1 is running.
<time 34> Process 1 is running.
<time 35> Process 1 is running.
<time 36> Process 1 is running.
<time 37> Process 1 is running.
<time 38> Process 1 is running.
<time 39> Process 1 is running.
<time 40> Process 1 is running.
<time 41> Process 1 is finished...
<time 41> All processes finished...
================================================================================
Average waiting time:       10.50
Average response time:      8.00
Average turnaround time:    17.33
Overall CPU usage:          100.00%
================================================================================
```



To separate the scheduling mechanisms from the scheduling policies, the main function was set up in scheduler.cpp to act as a driver function to call functions from it. This ensures that data flow starts from the main function and flows into the various other functions only to be processed or displayed. The beginning of the main function contains the command-line parser which also handles most of the error-handing functionality as well. The command-line parser was implemented mainly using argv[] which captures each spaced user input. argv[1] was used to collect the input file name, and argv[2] was used to collect the policy type. In the case of round robin, argv[3] was used to set the time quantum. In the example below, if policy_input (argv[2]) matched the string “FCFS,” the policy_type enum was set to FCFS and the task_loader sequence was started.

```
if (policy_input == "FCFS") {
    if (argc == 3) {
        policy = FCFS;
        std::cout << "Scheduling Policy: " << policy_input << "\n";
        task_loader(file_name, task_array, count);
        } else {
            std::cout << "Error: No time_quantum for FCFS.\n";
            }
```

After the name of the file is inputted into the task_loader function, the function checks if it is a valid file. The function then counts the number of tasks in the file and inputs each task into an array of type structure tasks. Each task contains fields for pid, arrival_time, and burst time. Later, start, finish, and remaining time is found. With count and task_array outputted from the task_loader function, the elements of task_array are pushed into the vector task_list for its high functionality. Finally, a switch case with policy_type enum policy is used to determine which set of functions will run.

```
int j = 0;
while (j<count) {
            task_list.push_back(task_array[j]);
            j++;
        }

switch (policy) {
    case FCFS:
        fcfs_policy(task_list, count, finish_array, wait_time);
        stats(count, finish_array, wait_time);
        break;
    case SRTF:
        srft_policy(task_list, count, finish_array, wait_time);
        stats(count, finish_array, wait_time);
        break;
    case RR:
        rr_policy(task_list, count, time_quantum, finish_array, wait_time);
        stats(count, finish_array, wait_time);
        break;
    default:
        break;
};
```

Each policy function takes task_list and counts them as inputs while outputting finish_array and wait_time. Starting with fcfs_policy, the clock variable and a ready queue that takes in tasks are created. The function works by iterating through task_list at each clock pulse to push any arriving or arrived tasks into the ready queue. Once complete, if the first task's remaining time is greater than 0, the remaining time is set to the burst time indicating that the process is now starting to access the CPU. This occurs until the remaining time is decremented eventually to 0 when the process is officially finished. The process is pushed to finish_array for its time values to be examined, and the ready queue is popped to allow the next task CPU access. Within the finish sequence, i is incremented to count the number of tasks completed. When i is equal to count, the while loop is exited. When no process is accessing the CPU and the ready queue is empty, wait_time is incremented to calculate CPU usage later.

```
void fcfs_policy(std::vector<task> &task_list, u_int &count, std::vector<task> &finish_array, u_int &wait_time) {
    u_int clock;
    std::queue<task> ready_queue;
    int i = 0;
    int k;

    while (i<count) {
        for (k=0; k<count; k++) {
                if (task_list[k].arrival_time <= clock && task_list[k].remaining_time == -1) {
                        task_list[k].remaining_time = task_list[k].burst_time;
                        ready_queue.push(task_list[k]);
                        } else continue;
                }
        if (!ready_queue.empty()) {
            if (ready_queue.front().remaining_time == 0) {
                ready_queue.front().finish_time = clock;
                finish_array.push_back(ready_queue.front());
                printf("<time %u> Process %u is finished...\n", clock, ready_queue.front().pid);
                i++;
                ready_queue.pop();
                continue;
            } else if (ready_queue.front().remaining_time > 0) {
                    if (ready_queue.front().remaining_time == ready_queue.front().burst_time) {
                        ready_queue.front().start_time = clock;
                        }
                        printf("<time %u> Process %u is running.\n", clock, ready_queue.front().pid);
                        ready_queue.front().remaining_time--;
                    }
                } else {
                    wait_time++;
                    printf("<time %u> No process running...\n", clock);
                }
                clock++;
            }

            printf("<time %u> All processes finished...\n", clock);
            printf("================================================================================\n");
        }
```

The next policy, SRTF used a similar approach to FCFS but utilized a vector of tasks to represent the ready_queue. This decision was made because the order of the processes in the queue does not matter as much in SRTF, only the amount of time the process has remaining. At each clock pulse, the arriving and arrived tasks were pushed into the ready queue. However, ready_queue is sorted in descending order using a Bubble sort algorithm for simplicity. With the queue sorted in descending order at each clock pulse, it ensures that the “back” element is the one with the lowest remaining time. The back process is run if it has the lowest remaining time. When the task’s remaining time is reduced to 0, the task is added to finish_array and the back of the ready queue is popped to allow other tasks access to the CPU.

```cpp
void srft_policy(std::vector<task> &task_list, u_int &count, std::vector<task> &finish_array, u_int &wait_time) {
    u_int clock;
    int i = 0;
    int k;
    int x;
    int y;
    std::vector<task> ready_queue;

    while (i<count) {
        for (k=0; k<count; k++) {
                if (task_list[k].arrival_time <= clock && task_list[k].remaining_time == -1) {
                        task_list[k].remaining_time = task_list[k].burst_time;
                        ready_queue.push_back(task_list[k]);
                        ready_queue.shrink_to_fit();
                        } else continue;
                }

                task temp;
                for (x=0; x<ready_queue.size(); x++) {
                    for (y=0; y<ready_queue.size()-x-1; y++) {
                        if (ready_queue[y].remaining_time < ready_queue[y+1].remaining_time) {
                            temp = ready_queue[y+1];
                            ready_queue[y+1] = ready_queue[y];
                            ready_queue[y] = temp;
                        }
                    }
                }
                if (!ready_queue.empty()) {
                    if (ready_queue.back().remaining_time == 0) {
                        ready_queue.back().finish_time = clock;
                        finish_array.push_back(ready_queue.back());
                        printf("<time %u> Process %u is finished...\n", clock, ready_queue.back().pid);
                        i++;
                        ready_queue.pop_back();
                        continue;
                    } else if (ready_queue.back().remaining_time > 0) {
                        if (ready_queue.back().remaining_time == ready_queue.back().burst_time) {
                            ready_queue.back().start_time = clock;
                        }
                        printf("<time %u> Process %u is running.\n", clock, ready_queue.back().pid);
                        ready_queue.back().remaining_time--;
                    }
                    } else {
                        wait_time++;
                        printf("<time %u> No process running...\n", clock);
                    }
                    clock++;
            }

            printf("<time %u> All processes finished...\n", clock);
            printf("================================================================================\n");

    }
```

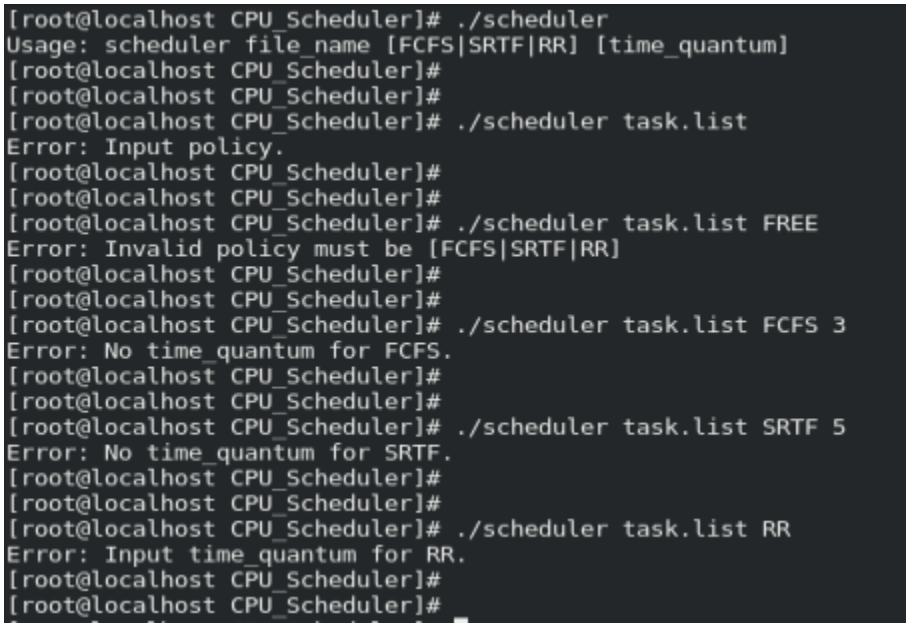
```cpp
void rr_policy(std::vector<task> &task_list, u_int &count, int &time_quantum, std::vector<task> &finish_array, u_int &wait_time) {
   u_int clock;
   std::queue<task> ready_queue;
   int i = 0;
   int k;
   int j = 0;

   while (i<count) {
       for (k=0; k<count; k++) {
               if (task_list[k].arrival_time <= clock && task_list[k].remaining_time == -1) {
                       task_list[k].remaining_time = task_list[k].burst_time;
                       ready_queue.push(task_list[k]);
                   } else continue;
               }
               if (!ready_queue.empty()) {
                   if (ready_queue.front().remaining_time == 0) {
                       ready_queue.front().finish_time = clock;
                       finish_array.push_back(ready_queue.front());
                       printf("<time %u> Process %u is finished...\n", clock, ready_queue.front().pid);
                       i++;
                       ready_queue.pop();
                       j=0;
                       continue;
                   } else if (ready_queue.front().remaining_time > 0) {
                       if (ready_queue.front().remaining_time == ready_queue.front().burst_time) {
                           ready_queue.front().start_time = clock;
                       }
                       if (j<time_quantum) {
                           printf("<time %u> Process %u is running.\n", clock, ready_queue.front().pid);
                           ready_queue.front().remaining_time--;
                           j++;
                           clock++;
                       } else {
                           j=0;
                           ready_queue.push(ready_queue.front());
                           ready_queue.pop();
                       }
                   }
               } else {
                   wait_time++;
                   printf("<time %u> No process running...\n", clock);
                   clock++;
               }
           }

           printf("<time %u> All processes finished...\n", clock);
           printf("================================================================================\n");
   }
```

```cpp
void stats(u_int &count, std::vector<task> &finish_array, u_int &wait_time) {
    float waiting_time, response_time, turnaround_time, cpu_usage = 0;

    int i;
    for (i=0; i<count; i++) {
        turnaround_time += finish_array[i].finish_time - finish_array[i].arrival_time;
        response_time += finish_array[i].start_time - finish_array[i].arrival_time;
        waiting_time += finish_array[i].finish_time - finish_array[i].arrival_time - finish_array[i].burst_time;
    }

    float avg_turnaround_time = turnaround_time / count;
    float avg_response_time = response_time / count;
    float avg_waiting_time = waiting_time / count;
    cpu_usage = 100 - wait_time;

    printf("Average waiting time:       %.2f\n", avg_waiting_time);
    printf("Average response time:      %.2f\n", avg_response_time);
    printf("Average turnaround time:    %.2f\n", avg_turnaround_time);
    printf("Overall CPU usage:          %.2f%\n", cpu_usage);
    printf("================================================================================\n");

}
```

The output from policy functions is finish_array and count. In stats.cpp, the average turnaround time was calculated by summing the difference between the finish and arrival times for each task and dividing that value by count. The average response time was calculated by summing the difference between the start and arrival times for each task and dividing that value

by count. Finally, the average waiting time was calculated by summing the difference between the finish time and both the burst and arrival times for each task and dividing that value by count.

Overall, I think my solution to this project is quite general in terms of how the user input is handled, the file data is read, and the stats are calculated. These parts of the project are largely scalable to large task list sizes. Adding policy is very simple as well, it only requires adding another case to the switch case and adding to the error handling sequence. The program also has great coverage of possible user input errors shown below with the output.



Finally, I think my code is elegant and simple to view and examine with good spacing and indentation. My solution I think is innovative as well. I used a switch case to act as the coordinator within the main function rather than create a separate function. Additionally, I enjoyed figuring out the algorithm for SRTF and I think my solution of using the vector as a queue was unique.