CS3104 P1 Scheduler

190020840

October 25 2022

1 Project Design and Datastructures

1.1 Polymorphism

The core code exists in the following files.

```
1    ./sched.c
2    ./fcfs.c
3    ./rr.c
4    ./p_rr.c
```

fcfs.c and rr.c contains the source code for a First Come First Serve (FCFS) and Round Robin (RR) scheduling algorithms. p_rr is a variant of the RR scheduler that runs the programs according to the provided priority (P_RR). The specification sheet suggested I start with a naive FCFS implementation and build off of the naive solution. However, one of the goals of the practical is to perform benchmarking and analysis on the scheduler I implement. In order to provide a reference point of the performance result of my implementation, I decided to keep the FCFS scheduler and allow a way for the user using the ./sched to choose the algorithm.

In such scenarios where we have multiple sets of functions serving a similar role, polymorphism would be exceptionally useful. However, C is not object-oriented so I had to simulate polymorphism in the following way by referencing [3]:

```
1  blocks *fcfs_load(pcb **pcbs, int pcb_count, char **
       ↪ args);
2  int fcfs_startup(blocks *b, int executed);
3  int fcfs_execute(blocks *b, int executed);
4  void fcfs_free_blocks(blocks *b);
5
6  blocks *rr_load(pcb **pcbs, int pcb_count, char **args
       ↪ );
7  int rr_startup(blocks *b, int executed);
8  int rr_execute(blocks *b, int executed);
9  void rr_free_blocks(blocks* b);
```

```
10
11 blocks *p_rr_load(pcb **pcbs, int pcb_count, char **
       ↪ args);
12
13 typedef struct scheduler {
14     blocks *(*loader)(pcb **, int, char **);
15     int (*starter)(blocks *, int);
16     int (*executor)(blocks *, int);
17     void (*free_blocks)(blocks*);
18 } scheduler;
19
20 int handle_args(scheduler *target_scheduler, int argc,
       ↪  char **argv);
```

*Excerpt from common.h

At runtime, the handle_args function determines the scheduling algorithm chosen by the user and dynamically allocates the relevant functions in the scheduler struct. I can then reference the loader, starter, executor, and free_blocks method without worrying about the underlying algorithm at all. This also allows for a flexible way to reuse code. The only difference between the priority variant of the RR scheduler with the RR scheduler itself is that the loader function is performing an insertion sort before running the program. This modular approach to developing the scheduler made it significantly easier to develop the priority variant by implementing an insertion sort for the loader function and reusing the starter, executor, and free_blocks methods.

1.2 Data Structures and Algorithms

1.2.1 Data Structures

Initially, I was planning on implementing a Multilevel feedback queue algorithm and identified that for polymorphism to work as expected, I would need to implement inheritance. This is because the dynamically chosen scheduler-specific methods need to have the same method signature but each scheduling algorithm would need different kinds of data structures to perform operations. The round-robin scheduling algorithm for example would need to store the quantum variable on top of the data structures necessary for the FCFS scheduler. The data structure for the Multilevel Feedback Queue algorithm would have to store pointers to the heads and tails of all the queues involved.

To resolve this issue, I created a wrapper class blocks that can store references to both fcfs_block and rr_block. The relevant methods allocated to the scheduler struct would then extract the relevant data structure and carry out its operation. Both the fcfs_block and rr_block are implemented as a Linked Lists. This approach fits neatly with the dynamic scheduler method selection as blocks can be reused (For example, the p_rr and rr schedulers can reuse the same rr_block data structure.)

```c
typedef struct fcfs_block {
    pcb *info;
    struct fcfs_block *next;
} fcfs_block;

typedef struct rr_block {
    pcb *info;
    struct rr_block *next;
    int quantum;
} rr_block;

typedef struct blocks {
    fcfs_block *fcfs_head;
    rr_block *rr_head;
} blocks;
```

*Excerpt from common.h

Regardless of the scheduling algorithm, they will all need to keep track of the variables such as the command and arguments as well as variables regarding execution times for benchmarking. These were all placed under the Process Control Block (pcb) struct.

```c
typedef struct {
    int process_id;
    int status;
    char *executable_path;
    char **arguments;
    int arg_count;
    int priority;
    char *full_line;
    int64_t begin;
    int64_t response_time;
    int64_t burst_time;
    int64_t turnaround_time;
    int64_t waiting_time;
} pcb;
```

*Excerpt from common.h

1.2.2 Methods

- loader – The loader's primary role is to encapsulate the pcb struct in the relevant data structure necessary to run the algorithm. The loader for RR will also assign the parsed quantum to the node at each recursion level. The loader for P_RR will perform insertion sort to build up the relevant data structure.
- starter – The specification sheet states that all processes should be started and immediately stopped using SIGSTOP and execute when necessary. The starter method for the relevant algorithm will perform this operation recursively over the relevant data structure for the chosen scheduler.
- executor – The FCFS executor will recursively iterate over the nodes in the Linked List and execute the command specified in the config file. It will wait until each command before moving on to the next node in the Linked List. The P_RR and RR executor will only run for the pre-defined quantum time and then move onto the next process. It will repeat this recursive operation until all the commands finish running.
- free_blocks – Recursively free the memory for the data structures used for the specific scheduling algorithm.

1.2.3 Algorithms

CPU scheduling decisions happen during the following four events [4]

1. Process switches from the running state to the waiting state
2. Process switches from the running state to the ready state
3. Process switches from the waiting state to the ready state
4. Process terminates

FCFS executor

```
1 def execute(fcfs_block head, int executed):
2     if(head is not null):
3         log execution start time
4         execute the process and wait until it
              ↪ terminates
5         log execution finish time
6         return execute(head.next, executed) + 1
7     else:
8         return 0
```

*Pseudo code for the executor for the FCFS scheduler

The FCFS scheduler is implemented as a non-preemptive scheduler that runs the next "ready" process only when the current process terminates, i.e. when events 1 and 4 are triggered. We can observe this behavior from Line 4 in the algorithm. This wait operation is performed by running waitpid without any flags (0) as the option:

```
1     int terminated = waitpid(current->info->process_id
          ↪ , &status, 0);
```

*Waiting for a process to finish in the FCFS scheduler

RR executor

```
1  def step(rr_block head, int executed):
2      if(head is not null):
3          if(process has not exited):
4              log execution start time
5              execute the process
6              wait for quantum microseconds
7              pause the process
8              if(process has exited):
9                  log execution FINISH time
10                 update status of pcb as completed
11                 return step(head.next, executed) + 1
12             else:
13                 log execution PAUSE time
14                 return step(head.next, executed)
15         else:
16             return step(head.next, executed) + 1
17     else:
18         return 0
19
20 def execute(rr_block head, int commands):
21     for(finished_execution = 0; finished_execution <
           ↪ commands ;;):
22         finished_execution = step(head, 0)
```

Pseudo code for the executor for the RR scheduler

An RR scheduler is a preemptive scheduler that runs around a list of processes in a circular fashion. [4] This operation is performed by using a loop on Lines 21 to 22. If we detect a process has terminated (whether it's at the current stack of recursion or during a previous iteration over the circular loop) we increment the number of completed processes and return that value to a higher-level stack. This way, we keep track of how many processes have been completed so we can stop our scheduling algorithm accordingly. These operations are handled on Lines 3, 8, 11, 14, and 16.

During each step of an RR scheduler, we run the program for a fixed predetermined time called a quantum, pause it, and then move on to the next step. [4] This operation is carried out on Lines 5 to 7.

The implementation side details of determining if a process has exited at a step of the RR scheduler are worth mentioning. man(3) waitpid [2] states :

wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.
waitpid(): on success, returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.
waitid(): returns 0 on success or if WNOHANG was specified and no child(ren) specified by id has yet changed state; on error, -1 is returned. Each of these calls sets errno to an appropriate value in the case of an error.

Excerpt from man(3) waitpid

If the return value of running waitpid with the WNOHANG flag equals 0, we know the child's status has not changed, i.e. the process still has not terminated. If the return value equals the pid of the child itself, we will know the child's execution status has been updated. This feature was used in the RR and P_RR scheduler to determine when the program has terminated.

2 Testing

2.0.1 Benchmarking Metrics

The following four metrics can help us get a grasp of the behavior of the scheduler [4] and were benchmarked for this practical

1. Throughput: Number of processes that are completed per time unit. At the end of the day, assuming prioritization is not involved, this is what the user of the operating system cares about the most over the long run.

2. Burst time: Total time the process is using the CPU. This will be helpful as a reference point when comparing the benchmarks of the FCFS and RR schedulers as we will get to see for how long a CPU is "held on to" for running the exact same processes. [1]

3. Turnaround Time: Total time required, from the process's perspective, for it to complete its execution. This is essentially burst time + waiting time and will indicate how long it took for the entire program to complete. There is a shortcoming however and that is this metric is tied to the speed of the output device

4. Waiting Time: Total time spent by the process waiting for access to the CPU. The scheduling algorithm directly affects the waiting time of processes as it is prioritizing the execution of the different processes.

5. Response Time: Time until the CPU gets its first access to the CPU. This metric is important for "short" processes. We can imagine a scenario where a process that needs 1 CPU clock cycle is running after a separate process that requires 100 clock cycles under a bad scheduler. In this case, the "throughput" will be negatively affected. Thus, response time is also a key component in measuring the effectiveness of a scheduling algorithm.

2.0.2 Results

The following configuration file was used for benchmarking in this section:

```
2 ./printchars a 40 b
15 ./printchars c 10 d
7 ./printchars e 50 f
1 ./printchars d 40 e
```

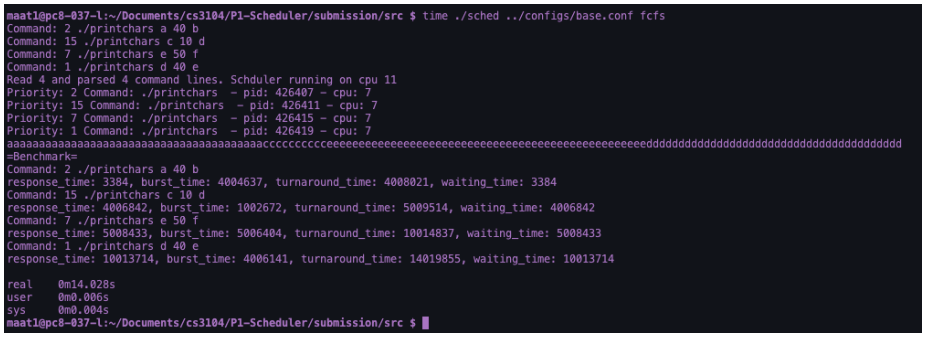

Fig 1. FCFS throughput = 4 * 60 / 14.028 = 17.10 processes per minute

Fig 2. RR quantum = 100 throughput = 3 * 60 / 9.021 = 47.84 processes per minute

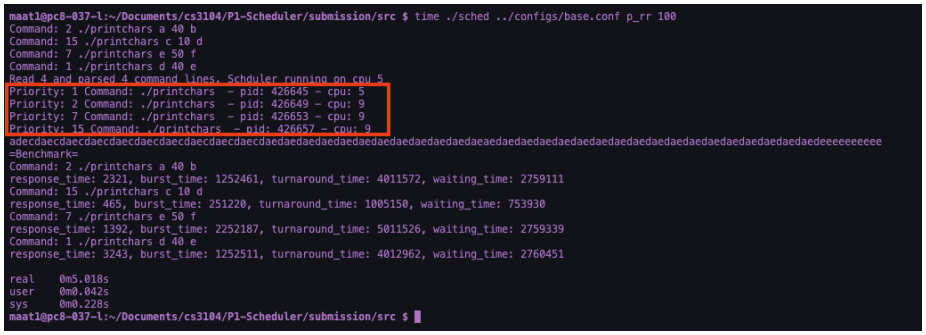

Fig 2. P_RR shows a similar throughput to RR. We observe priority of the command is respected

The following configuration file was used for benchmarking in this section:

```
2 ./printchars a 20 b
15 ./printchars c 30 d
7 ./printchars e 40 f
```

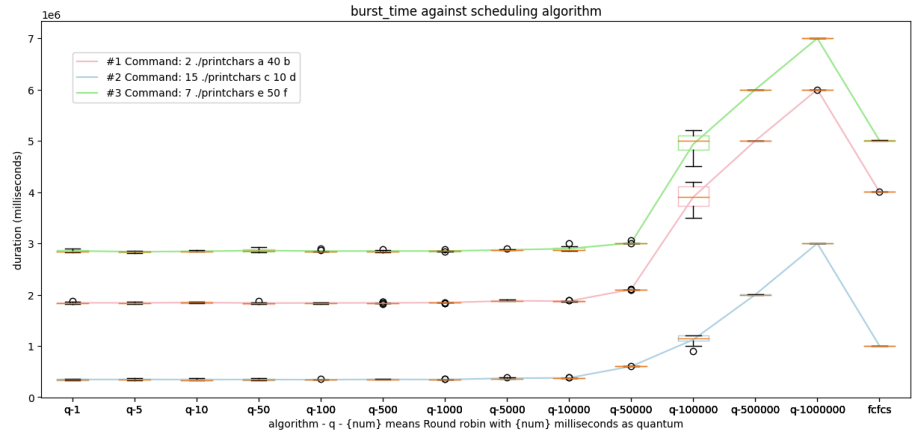

Fig 3. Scheduling algorithm against burst time

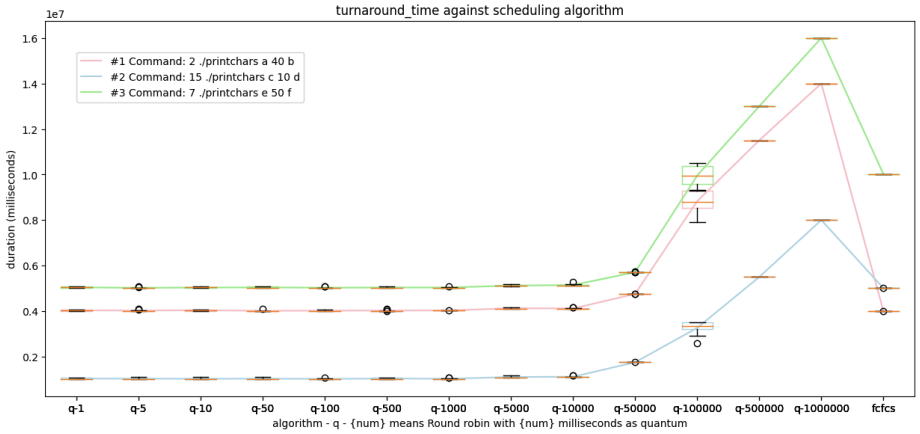

Fig 4. Scheduling algorithm against turnaround time

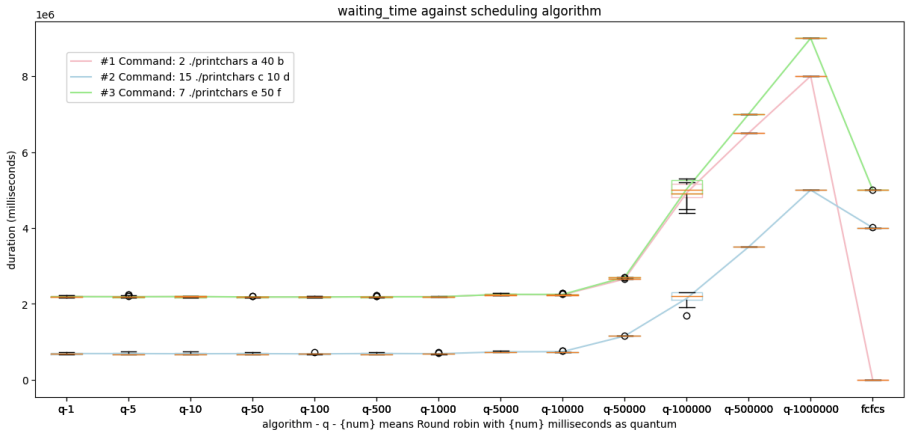

Fig 5. Scheduling algorithm waiting burst time

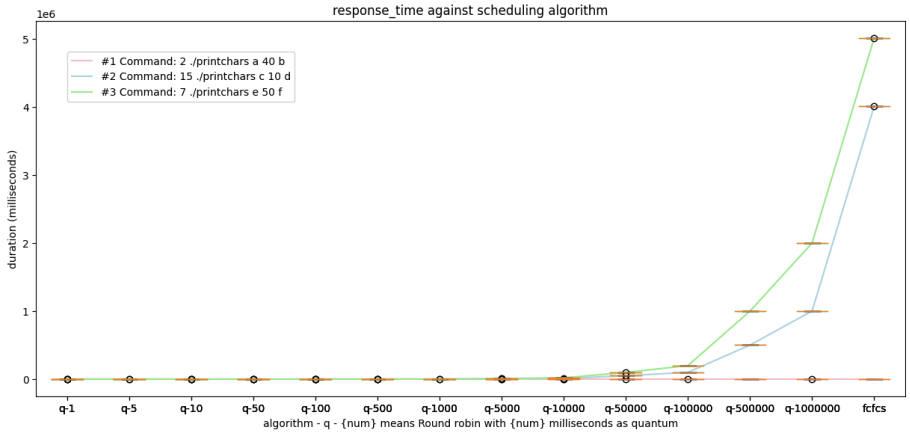

Fig 6. Scheduling algorithm against response time

2.0.3 Observations and Analysis

1. Fig 3: There is a positive correlation between the 2nd argument of ./printchars in the configuration file and the duration of the burst time for both the FCFS and RR schedulers. This is because the "actual work to be done", which is determined by the 2nd argument, is influencing how long the process is holding on to the CPU.

2. Fig 3: The burst time for the FCFS scheduler is higher than the RR schedulers with quantum $\leq$ 10000. This is because the ./printchars command is sleeping for most of its execution. FCFS still "holds on to" the CPU while it is executing a particular ./printchars command even though it is in a waiting state. However, RR moves on to the next process after the quantum period ends. This allows the scheduler to run the next ./printchars command while the current ./printchars command is executing.

3. Fig 4, 5: The waiting and response time of the first command in the configuration file is 0. The nature of the FCFS scheduler to run the first command in the queue, first, is causing this behavior.

3. Fig 4,5,6: We observe that the precedence of the command in the configuration file is correlated with the turnaround time, waiting time, and response time for the FCFS scheduler. The main issue causing this is the response_time. Because the latter commands in the configuration file have to wait for earlier commands to finish when running under the FCFS scheduler, the latter commands have a higher response_time which drives the waiting_time to increase, which thereby drives the turnaround time.

4. Fig 4,5,6: Unlike the FCFS, the RR scheduler turnaround time, waiting time, and response time is not correlated with the precedence of the command in the configuration file. Instead, it is correlated with the second argument in ./printchars. The RR scheduler circularly goes around the queue of processes and allows each of them to run. This is leading to a decrease in response time and waiting time since as a particular ./printchars command is waiting, another ./printchars is running its operation.

5. Fig 6: The response time for all the commands running under the RR scheduler is 0, at least until the quantum value reaches 10000. This is because each process quickly gets "access to the CPU" as its first turn comes around in the queue when running under the RR scheduler.

6. Fig 3,4,5: We observe that the RR scheduler performance degrades as the quantum value increases, to the point where the burst time, turnaround time, and waiting time is higher than the respective times for the FCFS scheduler. This issue happens when during a particular iteration over the circular queue, the running program finishes its execution but the RR scheduler's waiting period has not finished. This results in the CPU being idle even though there are programs that have yet to complete their execution.

2.0.4 Valid examples

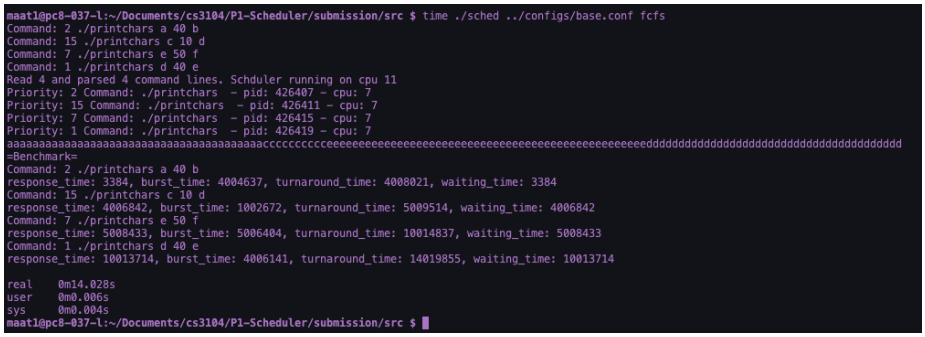

Fig 7. Valid FCFS execution

Fig 8. Valid RR execution

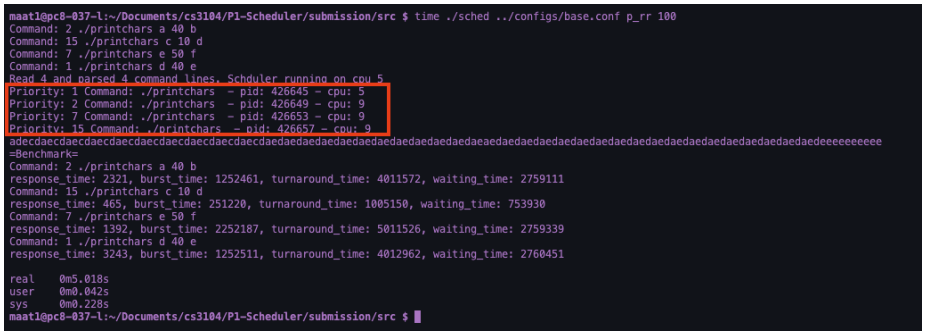

Fig 8. Valid P_RR execution

2.0.5 Invalid examples

Invalid ./sched configuration

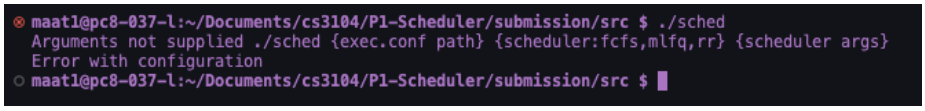

Fig 9. No Arguments supplied to the scheduler

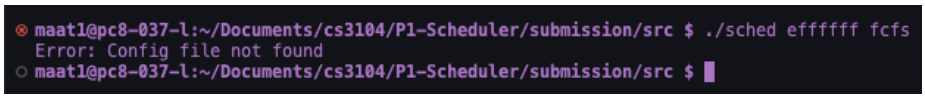

Fig 10. The configuration file cannot be found

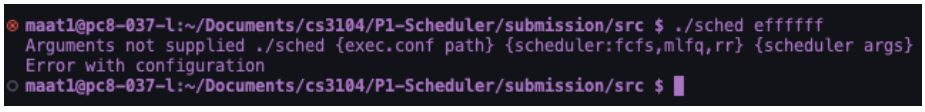

Fig 11. No scheduling algorithm provided as argument

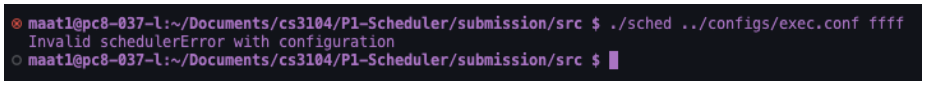

Fig 12. Unknown scheduling algorithm provided as argument

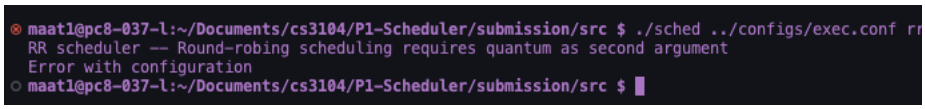

Fig 13. Round Robin Scheduler is chosen but the quantum is not provided

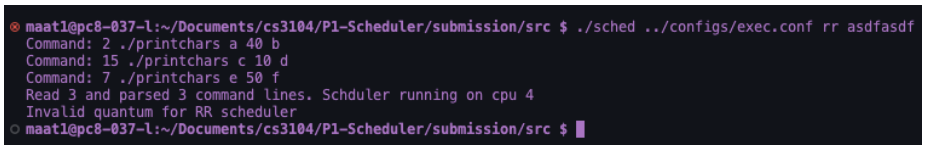

Fig 14. Unparsable Round Robin Scheduler quantum

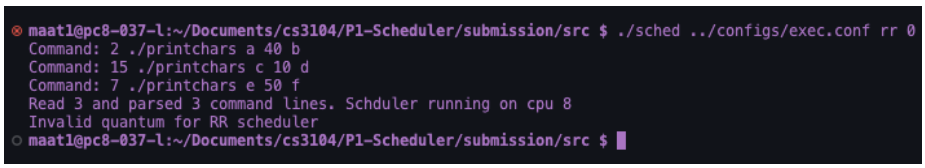

Fig 15. RR scheduler quantum should be greater than 0

Problems with configuration file

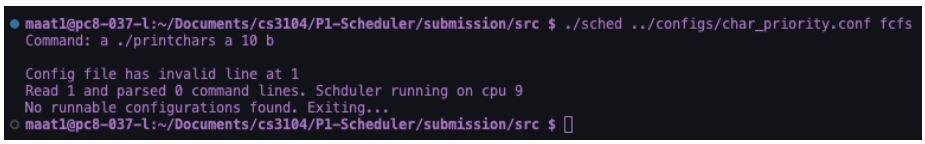

Fig 16. Unparasable priority value in scheduler

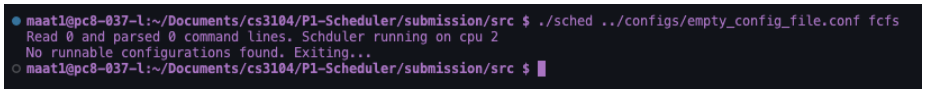

Fig 17. An empty configuration file is provided

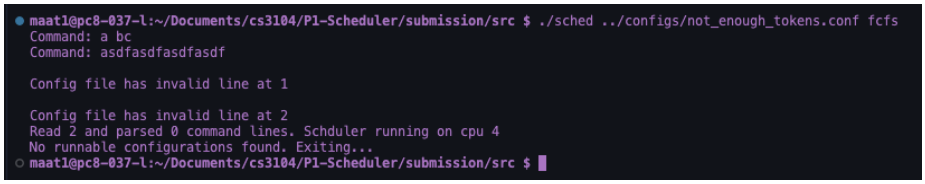

Fig 18. Invalid configuration file structure

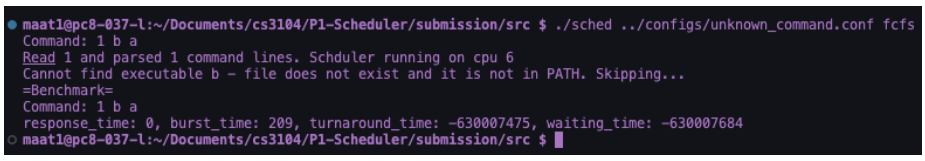

Fig 19. Command in the configuration file cannot be found in PATH

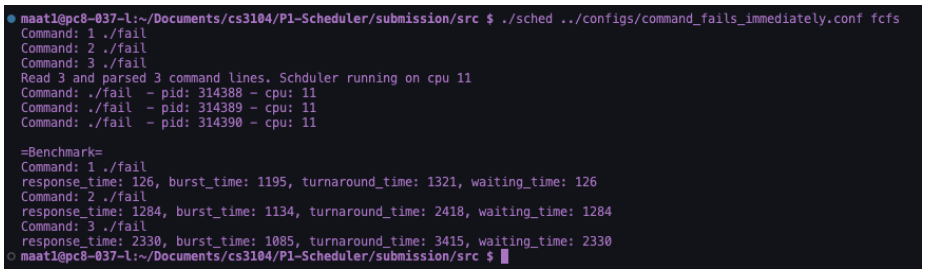

Fig 20. Command in configuration file fails immediately on launch – to account for situations where an incorrect set of arguments are provided

3 Additional Features

- Being able to choose a scheduling algorithm (Implemented by simulating polymorphism in C
- Being able to choose Round Robin scheduler quantum at runtime.

4 Development Process

4.0.1 Testing Strategy

To provide some context around the scheduling behavior metrics, the option to run the naive FCFS scheduler was kept. For each scheduling algorithm, (q-1, q-2,... q-1000000, fcfs), the scheduler was run 10 times to account for variations in the results. A series of box and whisker plots were then drawn for each scheduling algorithm against the burst time, turnaround time, waiting time, and response time to get a holistic understanding of how the respective scheduling algorithm and time quantum affect the behavior of the scheduling process. The effect of quantum on the P_RR scheduler was not tested as the executor for the RR scheduler was reused here, which would lead to similar results in the first place.

Edge case testing can be broken down into three components: 1. Invalid configuration of the ./sched program 2. Invalid configuration file 3. Problems with running the programs listed in the configuration file. The issues that might arise with these components were identified and duly handled.

4.0.2 Problems Faced During Development

1. Parsing the configuration file was by far the largest challenge. Being able to handle all the edge cases that might occur as well as dynamically allocating the necessary memory required to parse each line was a challenge.

2. Simulating polymorphism was another challenge. However, the initial investment in time made the code a lot more manageable.

5 Limitations

The main limitation with the current state of the problem is with the location from which the ./sched command is being run. The program assumes the user will run the program when it is in the same directory as the binary executable itself. The program also assumes the maximum length of each line in the configuration file is 1000 characters.

6 Conclusion and Summary

I have implemented a First-come-first-serve scheduler, a Round Robin Scheduler, and a priority-accounting Round Robin scheduler. I simulated polymorphism and inheritance in C to make the development process easier and more modular. I have then performed benchmarking on the throughput of each of these schedulers. I tested the impact of the burst time, turnaround time, waiting time, and response time for the FCFS scheduler and the RR scheduler. I have also observed the impact of how the quantum affects the performance of the RR scheduler.

6.1 References

References

[1] Scheduling of processes. http://www2.cs.uregina.ca/ hamilton/courses/330/notes/scheduling/scheduling.html.

[2] Waitpid(3) - linux man page. https://linux.die.net/man/3/waitpid.

[3] Axel-Tobias Schreiner. *Object-oriented Programming with Ansi-C*. 2011.

[4] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts 8th Edition Binder Ready Version*. Wiley Publishing, 8th edition, 2008.