OS Assignment-3

(Group-ID: 35)

Group Members:

S.No.	Name	Roll No.	Contribution
1	Debjit Banerji	2022146	• Researched on signals, MLFQ, and all other necessary topics • Wrote and debugged the program • Added all the comments in the program • Created and edited the explanation of the working of the SimpleScheduler.
2	Baljyot Singh Modi	2022133	• Researched on signals, MLFQ, and all other necessary topics • Wrote and debugged the program • Added all the error handling code snippets in the program

Working of SimpleScheduler:

1) Shell.c:
    a. This function is called by the main method and is responsible for taking the commands as input from the user through the terminal like interface.
    b. The program first checks whether it was able to read the command entered by the user properly. If yes, then the following steps are executed otherwise the program terminates with an error.
    c. The program then checks if the command entered is a valid command or not. If the command is valid then it formats and stores it in a 2D array for easy processing. If the command entered is found to be invalid, then the program asks for input from the user again.
    d. If the command entered begins with the word “submit”, then the process is added to the array in the shared memory and is made available to the scheduler, otherwise the process is executed normally by the shell and the shell asks for the next input from the user.

2) Scheduler.c:
    a. First the shared memory is setup and is linked to the one defined by the shell, then the variables store the values from the shared memory.

b. It also sets up the signals to be handled, the file descriptor for history.txt and the timer to imitate the “tslice”.
c. Now, the scheduler starts by waiting for “tslice” period of time and as soon as the “tslice” finishes, the scheduler checks for new processes to be added to the 4 priority queues. If there are new processes the processes are added (adding procedure described in the next step) or the scheduler waits for another “tslice” to receive processes and the pattern repeats.
d. If new processes are added and shared by the shell with the scheduler via shared memory, it creates new processes from the 2d arrays passed for the new processes, by setting the values of the fields of struct Process and also enqueuing it inside one of the priority queues.
e. If the priority number is mentioned and is between 1 and 4, the new process is added to the queue corresponding to the priority number specified. If the priority number is not specified by the user for a process or is not one of the 4 numbers (1,2,3,4), then the newly created process is added into priority queue 1.
f. After the new processes have been added to their respective queues, the round robin function is called which firstly calculates the minimum of the total number of processes and the number of processors available, and stores this number in a variable called “current_process_counter”.
g. After that it dequeues processes from the 4 priority queues, accessing each one of the 4 queues, till the time it gets adequate number of processes corresponding to the number stored in “current_process_counter”, and stores all the dequeued processes in a process array.
h. After this, the scheduler sets an alarm for imitating the “tslice” and launches each process. If the process is a new process (that is, the process has never been executed in the previous “tslices”) it redirects the control to the create_process_and_run2 which creates a new process using fork() and if the process is an existing process (that is, the process has been executed in at least one of the previous “tslices”), then the scheduler just sends a SIGCONT signal to the process to resume its execution from where it left in the previous tslice.
i. After the tslice is over, SIGALRM signal is invoked in the scheduler, which awakens it and then does the following tasks.
j. Firstly, the after the tslice is over the scheduler sends a SIGSTOP signal to all the processes that didn’t finish their execution in this “tslice” and then all of these processes get their execution time incremented (by the amount of “tslice”).
k. If there are any processes that do finish their execution during a particular “tslice”, then the scheduler catches the SIGCHLD signal sent by that process and registers all of its execution details like command name, execution time, waiting time, response time, turnaround time, etc. in the history.txt file.
l. Then, after sending the SIGSTOP signals to the processes, the rest of the processes in the 4 queues which didn’t get selected to be executed in this “tslice” gets their waiting time incremented (by the amount of “tslice”).
m. After these 2 steps, the scheduler then again checks if there are any new processes to be enqueued in the 4 queues.

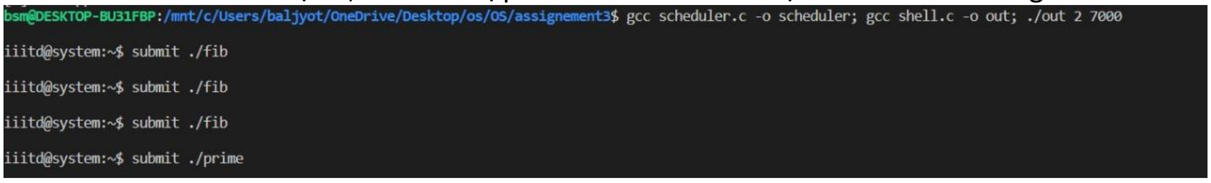
n. Then, the scheduler checks whether the number of “tslices” executed till now. If it’s equal to 5, then the scheduler increases the priority level of all remaining processes in the queue to level 1. Basically, after every 5 “tslices” this is done by the scheduler.
o. Then, the scheduler enqueues all the unfinished processes into the priority queue depending on the following criteria:
    i. If the number of “tslices” executed so far is 5, then all the unfinished processes are enqueued at the end of the priority queue 1. Basically, after every 5 “tslices” this is done by the scheduler.
    ii. If the number of “tslices” executed is not a multiple of 5, then the unfinished processes are enqueued at the end of lower priority, like for example, an unfinished process (initially at priority level 1) that ran in this tslice will be enqueued at the end of the priority level 2 queue. But if the priority level of the unfinished process is already 4 then it’ll be again enqueued in priority level 4 queue only.
p. After these steps, again round_robin function is executed and this cycle continues.

[**Note**: We have written the necessary code for “dummy_main.h” but are not using that since we have already handled that case in the scheduler itself, and we feel our implementation for that case in the scheduler is better and more efficient as it does not require us to create an additional file to be included in every test case.]

<u>Why is Priority queue better?</u>

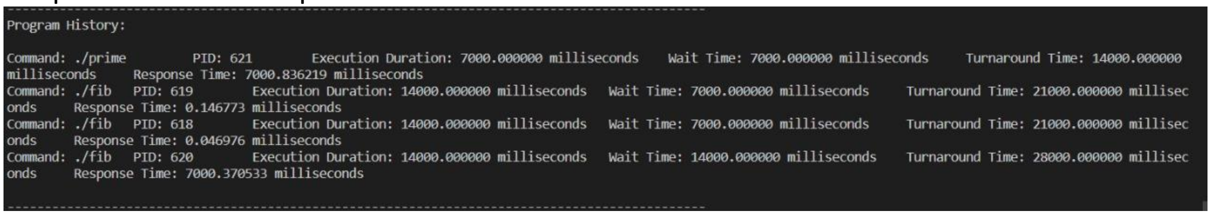
Now, to show how the SimpleScheduler implementation works better when using priority queues as compared to when not using priority queues, we have used the following test case:

In the below image, we have not used any priority scheduling, and we know that ./prime takes lesser time than ./fib, but here ./prime has to wait for ./fib to start running



```
bsm@DESKTOP-BU31FBP:/mnt/c/Users/baljyot/OneDrive/Desktop/os/OS/assignement3$ gcc scheduler.c -o scheduler; gcc shell.c -o out; ./out 2 7000

iiitd@system:~$ submit ./fib

iiitd@system:~$ submit ./fib

iiitd@system:~$ submit ./fib

iiitd@system:~$ submit ./prime
```

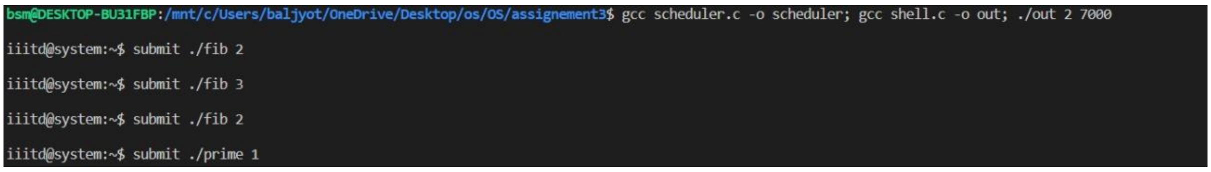
Output of the above inputs:



```
---------------------------------------------------------------------------------------------
Program History:

Command: ./prime        PID: 621        Execution Duration: 7000.000000 milliseconds    Wait Time: 7000.000000 milliseconds     Turnaround Time: 14000.000000
milliseconds     Response Time: 7000.836219 milliseconds
Command: ./fib   PID: 619       Execution Duration: 14000.000000 milliseconds   Wait Time: 7000.000000 milliseconds     Turnaround Time: 21000.000000 millisec
onds     Response Time: 0.146773 milliseconds
Command: ./fib   PID: 618       Execution Duration: 14000.000000 milliseconds   Wait Time: 7000.000000 milliseconds     Turnaround Time: 21000.000000 millisec
onds     Response Time: 0.046976 milliseconds
Command: ./fib   PID: 620       Execution Duration: 14000.000000 milliseconds   Wait Time: 14000.000000 milliseconds    Turnaround Time: 28000.000000 millisec
onds     Response Time: 7000.370533 milliseconds

---------------------------------------------------------------------------------------------
```

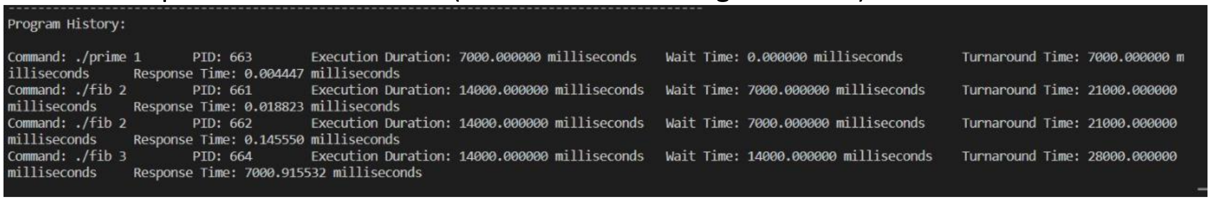
Here, the statistics of ./prime are as follows: (when run without using priority queues)
Response Time: 7000.836219 milliseconds
Waiting Time: 7000.000000 milliseconds
Execution Time: 7000.000000 milliseconds
Turnaround Time (Completion Time): 14000.000000 milliseconds

So, here priority queue becomes useful.
So, when we put ./prime as priority 1 and schedule all the ./fib programs at a lower priority level.



```
bsm@DESKTOP-BU31FBP:/mnt/c/Users/baljyot/OneDrive/Desktop/os/OS/assignement3$ gcc scheduler.c -o scheduler; gcc shell.c -o out; ./out 2 7000

iiitd@system:~$ submit ./fib 2

iiitd@system:~$ submit ./fib 3

iiitd@system:~$ submit ./fib 2

iiitd@system:~$ submit ./prime 1
```

Here after assigning lower priorities to all ./fib programs , we have given ./prime 1st priority. So, ./prime runs without having to wait for ./fib , thus it has lower response time and wait time as compared to the first case (where no scheduling was used).



```
Program History:

Command: ./prime 1      PID: 663        Execution Duration: 7000.000000 milliseconds    Wait Time: 0.000000 milliseconds       Turnaround Time: 7000.000000 m
illiseconds     Response Time: 0.004447 milliseconds
Command: ./fib 2        PID: 661        Execution Duration: 14000.000000 milliseconds   Wait Time: 7000.000000 milliseconds    Turnaround Time: 21000.000000
milliseconds    Response Time: 0.018823 milliseconds
Command: ./fib 2        PID: 662        Execution Duration: 14000.000000 milliseconds   Wait Time: 7000.000000 milliseconds    Turnaround Time: 21000.000000
milliseconds    Response Time: 0.145550 milliseconds
Command: ./fib 3        PID: 664        Execution Duration: 14000.000000 milliseconds   Wait Time: 14000.000000 milliseconds   Turnaround Time: 28000.000000
milliseconds    Response Time: 7000.915532 milliseconds
```

Here, the statistics of ./prime are as follows: (when run using priority queues)
Response Time: 0.004447 milliseconds
Waiting Time: 0.000000 milliseconds
Execution Time: 7000.000000 milliseconds
Turnaround Time (Completion Time): 7000.000000 milliseconds

GitHub Repository Link:
https://github.com/baljyot25/CSE231_OS_35/tree/main/OS_assignement_3