

P1 - A Scheduler in user space

Overview

In this practical, I implemented two types of schedulers: a round-robin scheduler and a priority scheduler. I achieved this by making use of a handful of system calls which allowed for processes to be created and executed. I also make use of signals and signal handlers which allowed for pre-emption of the processes where required. Finally, using self-defined structs, I implemented a process control block for each process along with the ready queue.

Design

This program is made up of three main components: the scheduling algorithms (in scheduler.c file), the process control block (in process.c and processStruct.h files) and the ready queue (in processList.c and processListStruct.h files). The ready queue contains all the processes in the order they were received, therefore it acts as a FIFO queue. Each process is stored in this queue as a process control block, which has all the necessary information needed for the scheduler to run the processes. The scheduling algorithm executes each process by going through the ready queue and accessing each process control block. Additionally, there is a priority queue, which is basically the original ready queue except it is ordered.

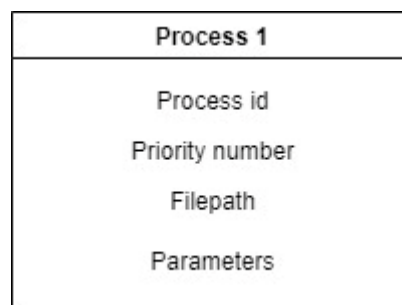


Figure 1: shows the all the data stored in the process control block for each process.

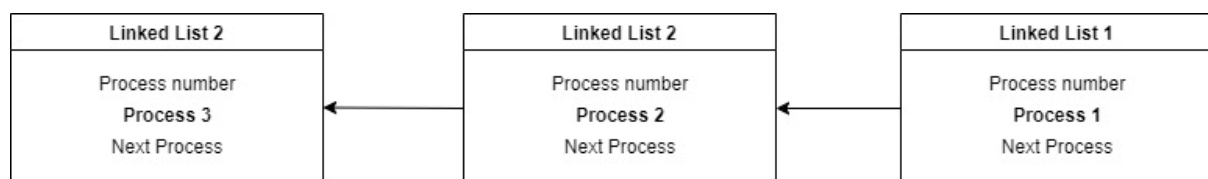


Figure 2: shows the general structure of the linked list.

Implementation

The process control block and ready queue were implemented using structs. Each process has its own struct which contains the process id, priority number, path of the program and the parameters to be passed to the program. This is contained in the process.c file. The structs of each of the processes is stored in a linked-list which implements the ready queue. This was done by having a struct that contains the process number, a struct variable which points to another linked-list struct and the struct of the process. To set up the linked-list each process struct is passed to the addProcess() function in processList.c which creates a linked-list struct for it to be stored. The linked-list struct which has the first process that is passed is assigned as the head of the linked-list and subsequent linked-list structs are added to the end of the linked-list. This is done using a loop to go through each linked-list struct in the linked-list until a null pointer is encountered, then the current linked-list struct which has the next process is added here. The following pseudocode show this functionality:

```
If head == null  
Then  
    head = new process  
Else  
    Last process = head  
    Repeat last process = next process in the linked list  
    Until next process = null  
    Next process = new process
```

Another functionality that is needed is for the scheduler to be able to get the process control block of the next process to execute. The nextProcess() method in processList.c does this. The linked-list is looped through until the next process is accessed which is then returned to the scheduler. To identify the correct process, an internal counter variable is compared with the process number variable in the linked-list struct to see if they match. The counter variable has an initial value of zero but when the first call to the function is done, it increments to one which indicates that the scheduler needs the first process from the linked list. When the counter has the value of the maximum number of processes, to avoid an out-of-bounds access, a conditional is used to assign the counter back to 0 whenever it reaches the maximum value and the whole process is repeated. So, this give an impression to the scheduler that the linked-list is circular despite it not actually being the case. The following pseudocode describes this functionality:

```
current process = head
Repeat
  If current process number == counter
  Then
    If current process number == total number of processes
    Then
      Counter = 0
    Return current process
  Else
    Get next process
Until current process == null
```

However, the above function only allows for the next process to be executed within a round-robin scheduler. This is because the algorithm doesn't take into consideration the priority of each process in the ready queue. Therefore, an additional function, `rearrangeList()` is included in the same file. This function takes each process from the initial linked-list and stores it in another ready queue where the processes are ordered in terms of their priority number. This queue is implemented as a simple array of process structs.

Both the scheduling algorithms are contained within the `scheduler.c` file where there are two main parts. The first part is adding each process to the ready queue. So, this deals with the transition between new and ready states. The second part is the actual scheduling algorithms which deals with the transition between the running and terminated states.

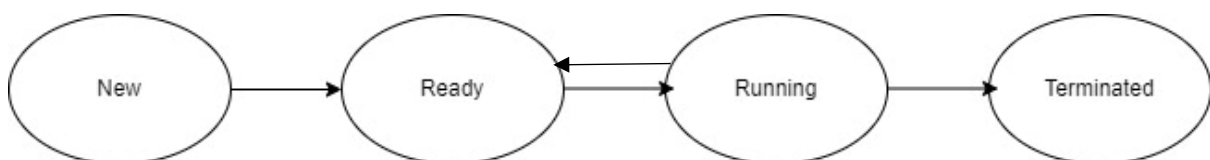


Figure 3: visual representation of the sequence of states the processes in the scheduling go through.

Since the process to executed are provided through a file, it is read line by line. For each line the `strtok()` function is used to split it into separate strings which are then stored in an array. These are the values to be added to the process control block, so they are passed to the process struct which is then passed to the linked-list struct so that the process is in the ready queue. Additionally, for each process control block that was read, a child process is created using `fork()` and then replaced by the intended process using `execl()`. But this process isn't executed and is instead immediately paused. This is done as a setup for when the scheduler starts executing each process. All these operations are done within a loop for every line read. Once all the above operations have been done, the ready queue is setup

with all the process control block for each process in the ready queue, so the scheduling algorithm can now start executing processes.

The user is then given an option to choose between the two schedulers. The first option is the round-robin scheduler. The general structure of this algorithm is a conditional statement within a loop. The loop is exited once the ready queue is empty. Otherwise, the `nextProcess()` function is called and each process is executed. The round-robin implementation is done simply by first calling a `kill()` function to send a `SIGCONT` signal. This causes the current process to transition from the ready state to the running state. Then the `usleep()` function is used to assign a time quantum which assigns a specified amount of time the process can run for. Once this time period is over, another `kill()` function is called and this time a `SIGSTOP` signal is sent. This causes the current process to move back to the ready state, but this time at the end of the FIFO queue. All of this is done if the current process that has been retrieved from the `nextProcess()` function hasn't finished execution, which is checked using the `waitpid()` function. If it has, then a `kill()` function is called and the `SIGTERM` signal is sent instead of a `SIGSTOP`. This causes the current process to transition from the running state to the terminated state. Following this, the queue is checked to see if it is empty. The following pseudocode describes this algorithm:

```
Repeat
    Get current process
    If current process has finished its execution
    Then
        Terminate it
        If queue is empty
        Then
            Empty = true
    Else
        Continue execution of the current process
        Pause the parent process
        Pause the execution of the current process
Until empty == true
```

The second option is the priority scheduler. The general structure for this scheduler is a simple loop. First, the `rearrangeList()` function is called to perform the calculations for reordering the processes. This new ready queue is retrieved by the scheduler and then in a loop, each process in the array is accessed, using an index number identifying each element in the array, and executed. Execution happens non-preemptively since this is how priority schedulers usually operate. As a result, the parent process starts the execution using `kill()` and then waits until the child process has finished executing using the `wait()` system call. After the child process has finished executing, another `kill()` function is called. This algorithm is described using the following pseudocode:

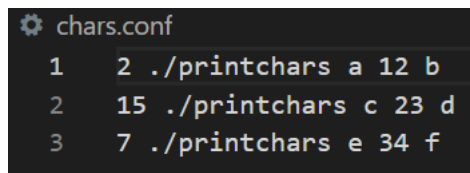
```
call rearrangeList()
Get priority queue
Repeat
    Get next process
    Start execution of child process
    Wait until child process finishes
    Terminate child process
Until ready queue empty == true
```

Testing

I tested my program by using the configuration file provided initially. After my program executed each process from that file correctly, I modified the file by adding more processes with different attributes for the scheduler to handle. This was to check whether the schedulers can successfully execute under varying situations, therefore ensuring the robustness of code. I did this for each scheduler.

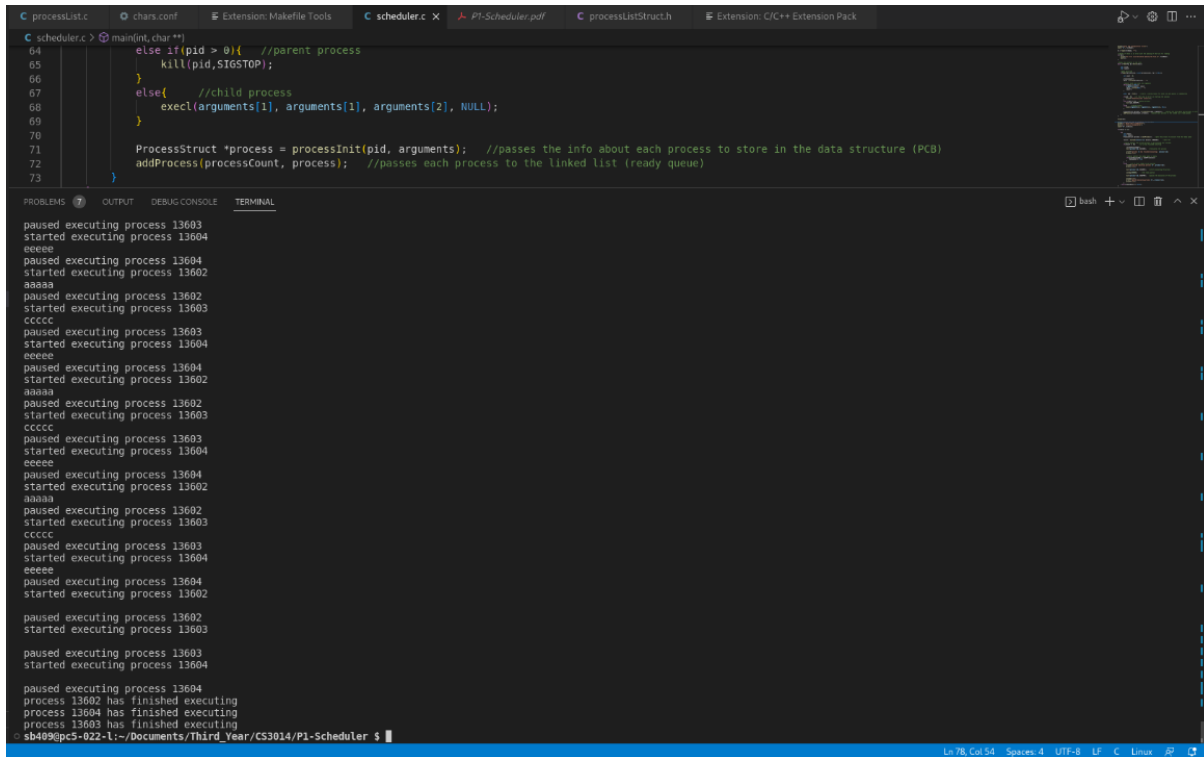
For the round-scheduler,

- The first test case checks that the original configuration file is read, and that the processes are executed correctly.
 - Input: original configuration file.



```
chars.conf
1 2 ./printchars a 12 b
2 15 ./printchars c 23 d
3 7 ./printchars e 34 f
```

- Output: Correct output messages and correctly executed processes where each process executed for a certain time and is then swapped.

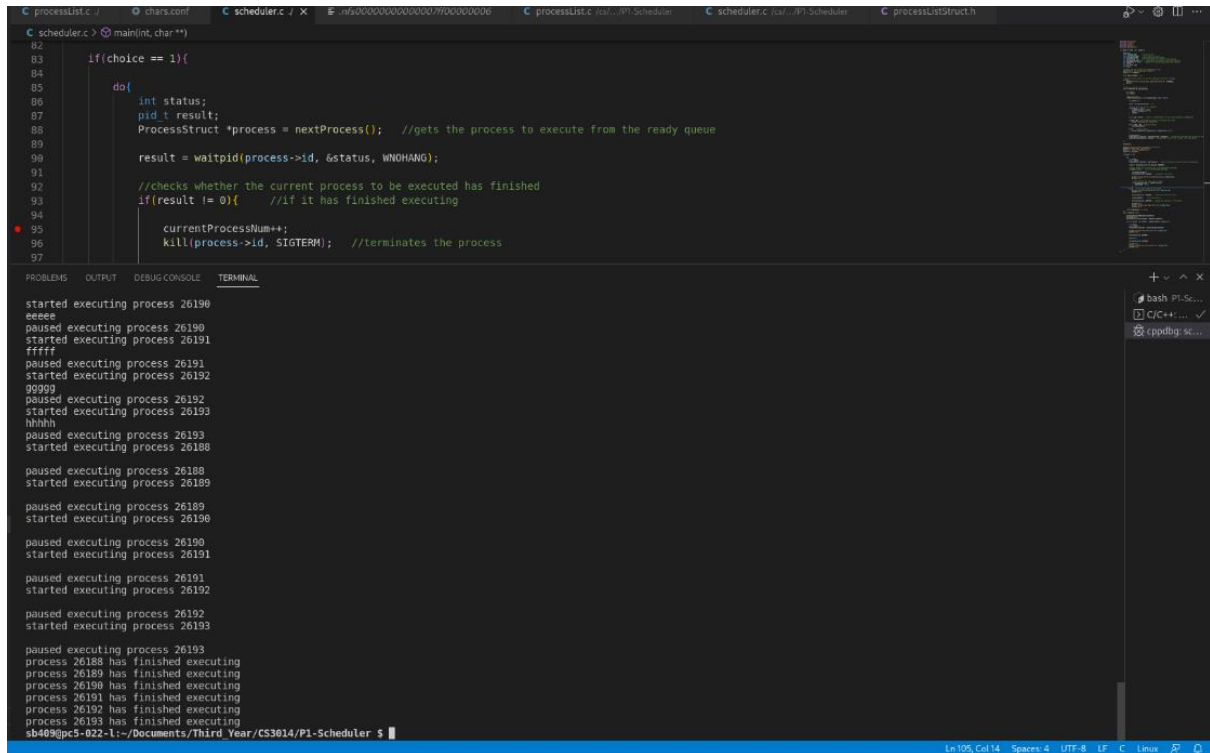


The screenshot shows a VS Code editor with a C file named `process.c` and a terminal window. The C code implements a scheduler with a `main` function that takes `argc` and `argv` as arguments. It uses `fork` to create child processes and `wait` to manage their execution. The terminal output shows the execution of three processes (13602, 13603, 13604) with their respective outputs: `aaaa`, `cccc`, and `eeee`. The terminal also shows the completion of each process and the final state of the scheduler.

- The second test case checks that the scheduler can handle even more processes.
 - Input: the modified configuration file

```
chars.conf
1 2 ./printchars a 12 b
2 15 ./printchars c 23 d
3 7 ./printchars e 34 f
4 3 ./printchars g 23 d
5 4 ./printchars h 34 f
```

- Output: Correct output messages for each process and all the process executed correctly

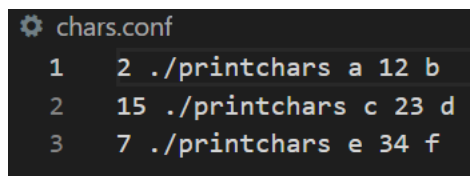


The screenshot shows a C++ IDE with a file named `scheduler.c` open. The code is a priority scheduler implementation. It uses a `ProcessStruct` to represent processes, with fields for `id`, `priority`, `time`, and `status`. The `main` function starts by reading a configuration file `chars.conf` and then enters a loop where it repeatedly selects the process with the highest priority from a ready queue, executes it for a specified time, and then swaps it with the next process in the queue. The terminal output shows the execution of several processes, each with a unique ID and a specific execution time. The output also indicates when a process has finished executing.

```
1 2 ./printchars a 12 b
2 15 ./printchars c 23 d
3 7 ./printchars e 34 f
```

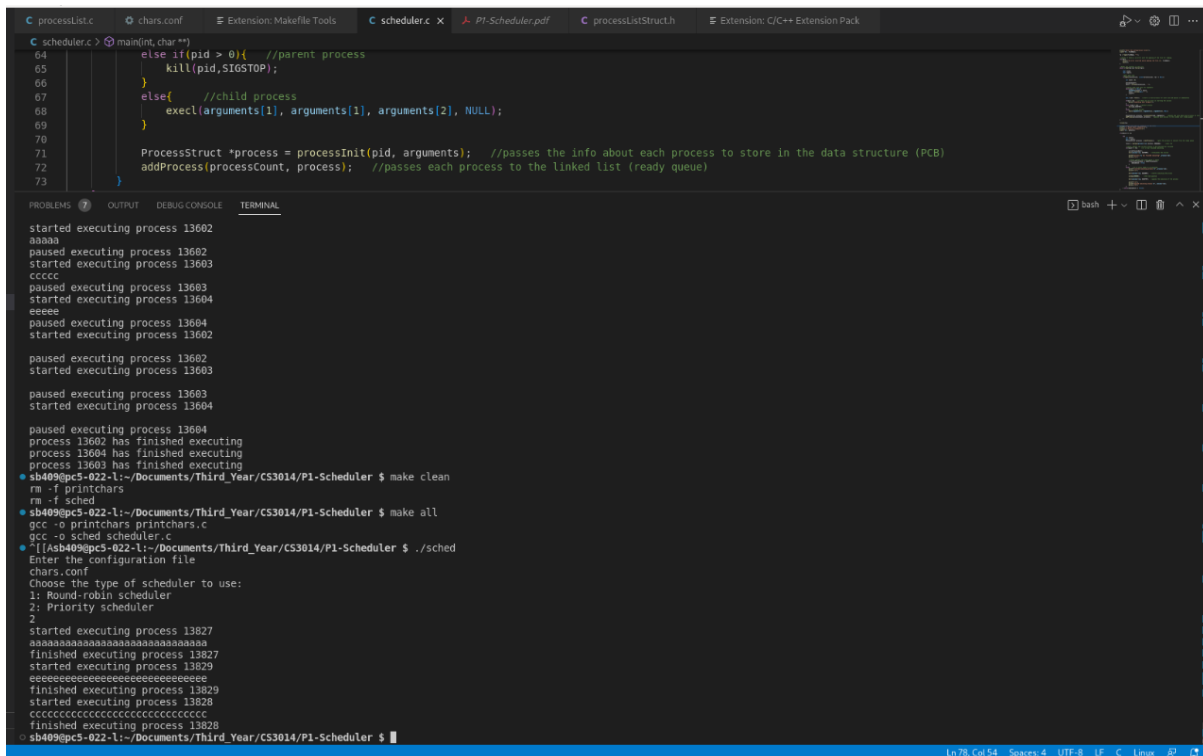
For the priority scheduler,

- The first test case also checks that the original configuration file is read, and that the processes are executed correctly.
 - Input: original configuration file



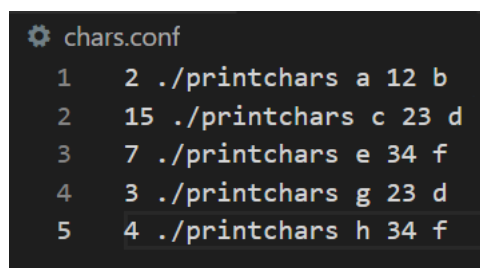
The screenshot shows the `chars.conf` configuration file. It contains three lines of configuration data, each representing a process to be executed. The first line is `1 2 ./printchars a 12 b`, the second is `2 15 ./printchars c 23 d`, and the third is `3 7 ./printchars e 34 f`.

- Output: Correct output messages and correctly executed processes where each process executed for a certain time and is then swapped.



The screenshot shows a VS Code editor with a C++ scheduler program. The code in `scheduler.c` includes a `main` function that takes command-line arguments and uses `fork` to create child processes. It uses `ProcessStruct` to manage processes and a linked list to store them. The terminal output shows the execution of the program, including the creation and execution of multiple processes (e.g., 13602, 13603, 13604) and the completion of the scheduler. The user also runs `make clean` and `make all` to compile the program.

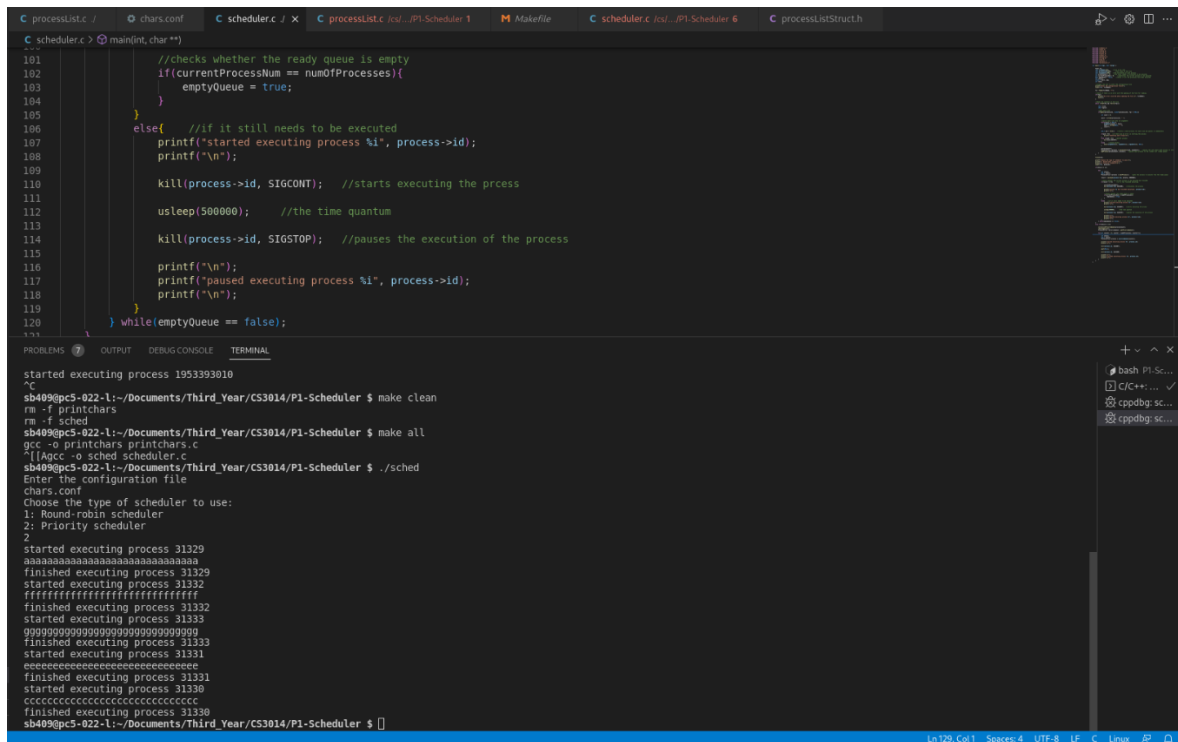
- The second test case also checks that the scheduler can handle even more processes.
 - Input: the modified configuration file



The screenshot shows the `chars.conf` configuration file. It contains five lines of configuration, each specifying a process ID, a command to run, and two additional parameters.

```
1 2 ./printchars a 12 b
2 15 ./printchars c 23 d
3 7 ./printchars e 34 f
4 3 ./printchars g 23 d
5 4 ./printchars h 34 f
```

- Output: Correct output messages for each process and all the process executed correctly



```
101 //checks whether the ready queue is empty
102 if(currentProcessNum == numOfProcesses){
103     emptyQueue = true;
104 }
105 }
106 else{ //if it still needs to be executed
107     printf("started executing process %i", process->id);
108     printf("\n");
109     kill(process->id, SIGCONT); //starts executing the process
110     usleep(500000); //the time quantum
111     kill(process->id, SIGSTOP); //pauses the execution of the process
112     printf("\n");
113     printf("paused executing process %i", process->id);
114     printf("\n");
115 }
116 } while(emptyQueue == false);
```

```
started executing process 1953393010
^C
sb409@pc5-022-l:~/Documents/Third_Year/CS3014/P1-Scheduler $ make clean
rm -f printchars
rm -f sched
sb409@pc5-022-l:~/Documents/Third_Year/CS3014/P1-Scheduler $ make all
gcc -o printchars printchars.c
[[Apcc->0 sched scheduler.c
sb409@pc5-022-l:~/Documents/Third_Year/CS3014/P1-Scheduler $ ./sched
Enter the configuration file
chars.conf
Choose the type of scheduler to use:
1: Round-robin scheduler
2: Priority scheduler
2
started executing process 31329
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
finished executing process 31329
started executing process 31332
fffffffffffffffffffffffffffff
finished executing process 31332
started executing process 31333
ggggggggggggggggggggggggggggg
finished executing process 31333
started executing process 31331
eeeeeeeeeeeeeeeeeeeeeeeeeeee
finished executing process 31331
started executing process 31330
ccccccccccccccccccccccccccc
finished executing process 31330
sb409@pc5-022-l:~/Documents/Third_Year/CS3014/P1-Scheduler $
```

One thing that needs to be pointed out is that after the processes have finished executing, the program doesn't go straight to displaying a message saying that the process has been terminated. But instead, it goes through the loop once more without producing any output string from the process before displaying that output.

During the development of the schedulers, there were numerous errors that were brought up along the way. This was due to an insufficient amount of testing as the program was being developed. The result of this was unnecessarily long periods of debugging which really hindered the development process. Most of the errors were regarding memory, as I was got numerous segmentation faults.

Evaluation

After making sure that the program works correctly, I analyzed the performance of both the schedulers. I did this by adding a timer to the program, where the timer was started and stopped at specific points, and the amount of time is calculated by subtracting both values. The starting point and stopping points were placed at appropriate places such that the

required calculations could be made. I decided to calculate the turnaround time and waiting time for each process and then working out the average. The initial configuration file was used when performing the calculations.

For the round-robin scheduler,

- The average turnaround time:
 - $(1058 + 1056 + 1092) / 3 = \underline{1069 \text{ clock ticks}}$
- The average waiting time:
 - $(206 + 205 + 215) / 3 = 209 \text{ clock ticks}$ (this is the average time for a process to run without interruption)
 - $1069 - 209 = \underline{860 \text{ clock ticks}}$ (this is achieved by subtracting the average time to run uninterrupted from the average turnaround time)

For the priority scheduler,

- The average turnaround time:
 - $(107 + 110 + 138) / 3 = \underline{118 \text{ clock ticks}}$
- The average waiting time:
 - $(0 + 107 + 217) / 3 = \underline{108 \text{ clock ticks}}$

From the results above, it is evident that the priority scheduler had better performance as each process was able to execute quicker and wait less.

Conclusion

Using my knowledge of scheduling in operating systems as well as my knowledge in systems programming in C, I was able to develop two user-level schedulers. Through numerous periods of testing and debugging, I was able to show that my program can execute correctly and achieve desirable results.

However, there are a few limitations in my implementation. The most obvious one is that the schedulers are only able to execute processes that take a single input. Another limitation would be that there were too few test cases which don't necessarily show robustness of the code.

There are few things I could have done better to not only ease the development process but to also produce a better program. Firstly, should have tested by code more by adopting a more test-drive. This would allow me that my program can in any provided situation. Secondly, some parts of the code could have been more efficient. For example, the `rearrangeList()` function is unnecessarily complicated. Additionally, the overall code could have been cleaner and more elegant, so this is something I should also work on.

In conclusion, my approach has both some strengths and some weaknesses which are summarized below:

- Strengths:
 - Storing processes in structs make it easy to access the process attributes
 - Linked-list implementation allows each process to be retrieved easily
 - Simple loop structure to execute each code
- Weakness:
 - Only accepts one parameter per process
 - Limited testing
 - Inefficient and inelegant parts of code