Introduction:

In this practical, we had to implement various scheduling algorithms using signals and system calls in the C programming language. The task was to develop a simple user space scheduler that would execute and arbitrary list of other programs and control them by sending them signals to ensure that only one of these programs is running at any given time. We had some flexibility since the algorithm and data structure choices were left to us.
Our program had to follow these guidelines :

- *Your program, once compiled, should be contained in a binary named sched.*
- *It should accept a single command line parameter: path to a configuration file which contains*
- *a list of programs in the format specified below.*
- *It should execute each of the listed programs by using a combination of fork() and exec()1.*
- *All processes should be stopped by the parent immediately after being launched, by sending them the SIGSTOP signal.*
- *Information about each of the processes should be recorded in a data structure (consisting of*
  Process Control Blocks) which contains relevant information (e.g. the path to the executable, parameters and process ID/pid).
- *The parent process should then process the data structure containing process control blocks and activate one process at a time by sending it the SIGCONT signal.*

Design and implementation:

basic submission:

When I came to implementation and design for the basic submission of the practical. We followed the approach described in the specification sheet.
After making sure that the skeleton was working correctly and that we could call the appropriate program using hard-coded fork()'s and exec()'s, we got into the first bit on development.
We first had to create a data structure to hold information about the processes. We used a simple linked list of structs, where each struct contains the ID of the process returned by fork(), the path to its executable, priority, and any parameters:

```
struct process {
    pid_t pid;
    int priority;
    char* path;
    char* parameters;
    struct process *next;
};
```

Then, we had to read each line in the configuration file and populate the data structure above.
To do so, We coded a function with the following prototype
~ `struct process* fileReader(char *filename , struct process* head);` ~
After opening the file in read-mode, We loop through the file line by line using a while loop. We then use strtok() to tokenize our elements as described in the specification sheet :
<priority> <path> <arguments>

We malloc the tokenized strings and strcpy() them so that the char pointers in our actually point to the right elements. Then we use the add() method that we coded to add these parameters to the linked list.

We have some methods that allow us to manipulate our linked list easily, add() adds elements to the list, count() returns the amount of elements in the list, and pront() prints the full linked list.

Now that we are able to populate our data structure with the right process values, we can write a simple FIFO/FCFS batch scheduler which waits for process to finish before moving on.
This part is the commented bit in my main method.
The first thing we do before thinking about the scheduling algorithm is find a way to create and store our processes in a clean and simple way.
We do so using a while loop and iteration through the linked list where we just stored all the line of program command that will be called later. Thus, we immediately create a process and stop the parent at launch. The we call the execl() using the following arguments :
execl(current_node->path, current_node->path, current_node->parameters, NULL);

we also store the process id in the struct with the following:
current_node->pid = propid;

We do so for every element in our data structure. Then we get to the scheduling algorithm.
Using the pid field in our struct to select the process that we want to handle, we iterate once again through the struct and use the following sequence :

- *"int status;" and "pid_t result = waitpid(other_node->pid, &status, WNOHANG);" will allow us to wait for the process to end before moving on.*
- *"kill(other_node->pid, SIGCONT);" this kill call starts the process execution.*
- *"while (waitpid(other_node->pid, &status, 0) != -1)" We will stay in this loop until the process is done.*
- *We then call a SIGSTOP and a SIGTERM to clean at the end of the loop.*

Arriving at the fourth and final step of the basic submission design and implementation, we had to turn this into a round-robin scheduler with a fixed quantum by waking a process, sleeping for the duration of the quantum, and then stopping the process and moving to the next one.
Here, we iterate again through the linked list. Keep in mind that we have created an integer called "i" that we will use to determine if all the processes are done. The first big thing we do in our loop is to check if the process with the corresponding id is running, we do so with the following :
if(kill(other_node->pid,0) == 0)

When getting into this conditional, we use SIGCONT and SIGSTOP in combination with usleep(q) where q is the quantum. Let's not forget that calling kill on a process wakes it up.
If the process is not alive any-more, we increment "i" by 1. Then when it's time to go to the next node at the end of the loop, if the next node is null and I is not yet equal to the amount of elements in our data structure, we reset the list's head. Else, we just go to the next node normally.

This allows us to have a round-Robin schedule algorithm.

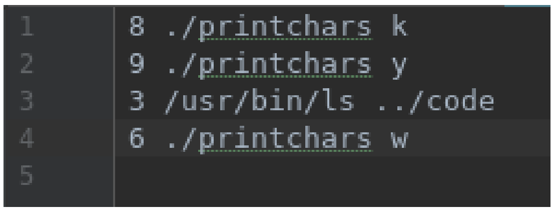
Extension:

I achieved the priority Queue extension. I put it in the priority.c file and it can be run using
./prio <path>.

Here, The list is so created so that the highest priority element is always at the head of the list. The list is arranged in descending order of elements based on their priority. This allow us to remove the highest priority element when using the push() method. We traverse the list and find the proper position to add the element so that the priority is preserved. Thus, the push() operation takes O(N) time.

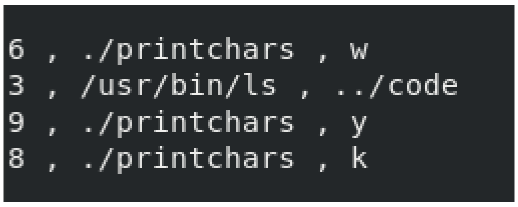
Testing:

First and foremost, We had to make sure that the content of our data structure is actually valid. This is our config file :



```
8 ./printchars k
9 ./printchars y
3 /usr/bin/ls ../code
6 ./printchars w
```

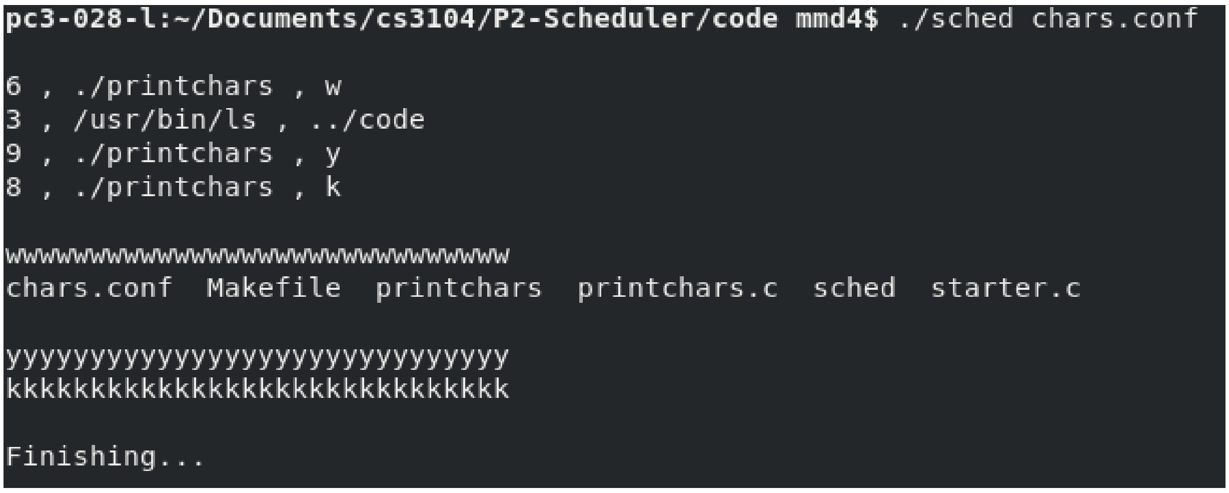
And this is the content of our data structure that we get using the pront() method.



```
6 , ./printchars , w
3 , /usr/bin/ls , ../code
9 , ./printchars , y
8 , ./printchars , k
```

Now we know that our linked list is valid (I passed multiple test to make sure that it would be fine, this is one of many examples).

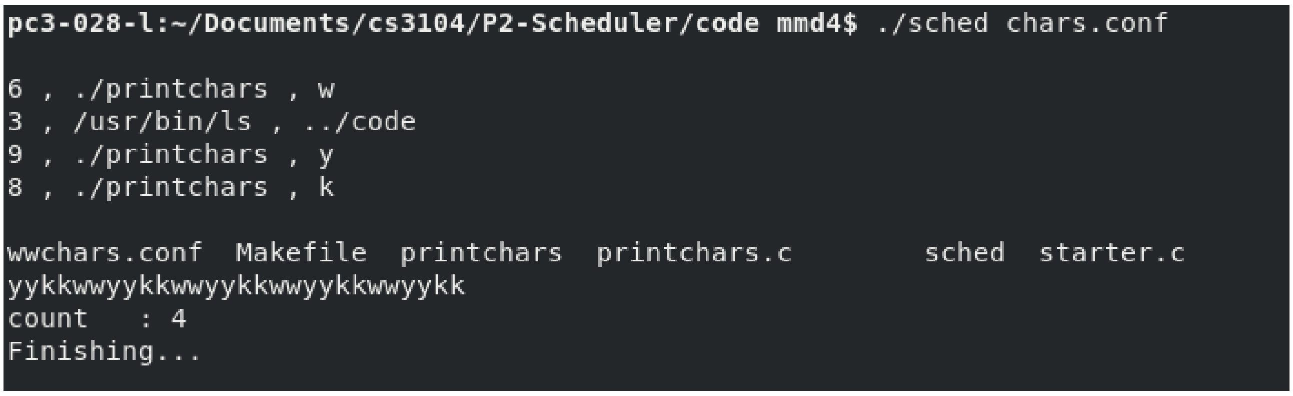
Now let’s check that our FIFO implementation works properly, using the same config file, we should get a succession of 30 “w” printed, then the content of the code directory and 30 “y” and “k”.   This is the terminal output:



```
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./sched chars.conf

6 , ./printchars , w
3 , /usr/bin/ls , ../code
9 , ./printchars , y
8 , ./printchars , k

wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
chars.conf  Makefile  printchars  printchars.c  sched  starter.c

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkk

Finishing...
```

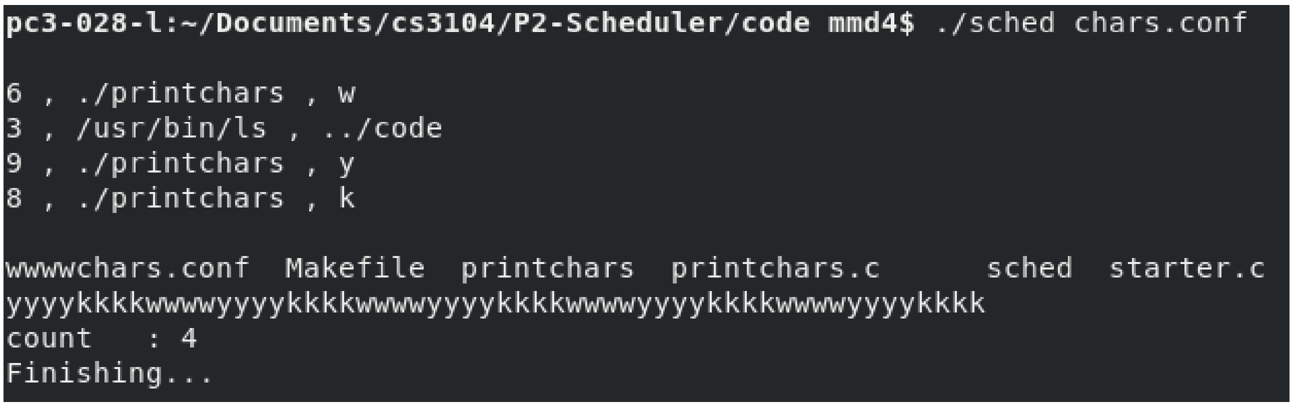
As we can clearly see, The FIFO implementation work perfectly.

Now we should check the Round-Robin algorithm!
Using again the same config file and using a low Quantum, we should get a small amount of “w” then the ls call in one block and then a loop of the same small amount of “y” and “k” starting with “w” again.



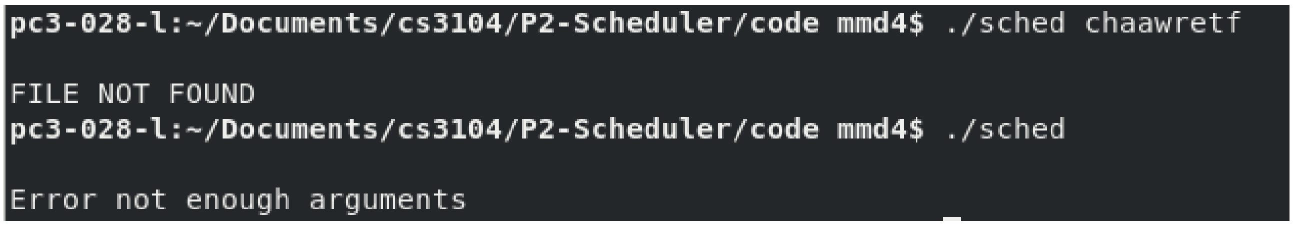
```
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./sched chars.conf

6 , ./printchars , w
3 , /usr/bin/ls , ../code
9 , ./printchars , y
8 , ./printchars , k

wwchars.conf  Makefile  printchars  printchars.c        sched  starter.c
yykkwwyykkwwyykkwwyykkwwyykk
count   : 4
Finishing...
```

As we can see here, we have the predicted result which proves that the algorithm works, but this is not a full proof. We have to check if having a higher quantum actually makes the amount of chaining letters more important. Thus, we increase the quantum and check what happens:



```
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./sched chars.conf

6 , ./printchars , w
3 , /usr/bin/ls , ../code
9 , ./printchars , y
8 , ./printchars , k

wwwwchars.conf  Makefile  printchars  printchars.c      sched  starter.c
yyyykkkkwwwwyyyykkkkwwwwyyyykkkkwwwwyyyykkkkwwwwyyyykkkk
count   : 4
Finishing...
```

We have the correct output. The Round-Robin Works.
Now let’s see at some error checking to see if our code is full-proof.



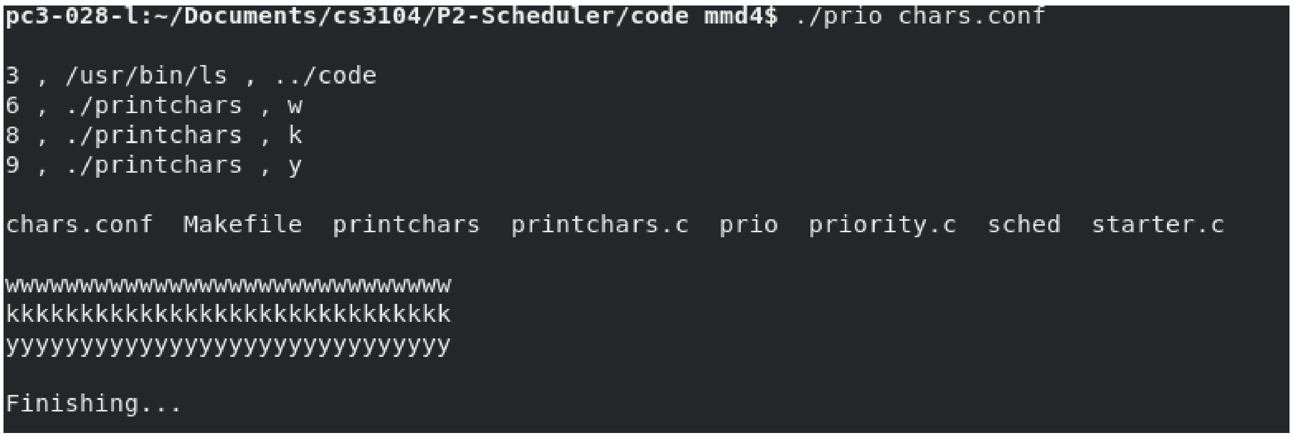
```
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./sched chaawretf

FILE NOT FOUND
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./sched

Error not enough arguments
```

The error checking concerning the argument and the file seems to work. Plus we added some error checking when reading the file to handle null characters and other corrupted data using conditionals such as :

```
if(par_token == NULL){
   par_token = "";
}
```

Extension testing:

Using the same config file again, We notice that the priorities are not in the right order. This is the output of the extension program:



```
pc3-028-l:~/Documents/cs3104/P2-Scheduler/code mmd4$ ./prio chars.conf

3 , /usr/bin/ls , ../code
6 , ./printchars , w
8 , ./printchars , k
9 , ./printchars , y

chars.conf  Makefile  printchars  printchars.c  prio  priority.c  sched  starter.c

wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

Finishing...
```

We notice that the priorities are in the right order in the Linked list printing. Furthermore, The commands execute according to their priority!
We can conclude that this is a success.

Evaluation and conclusion:

In this practical, I really focused on having a clean code and a clean report while still going beyond the original specification by implementing an extension. My report is also pretty well written. Focusing on these aspects and on simplicity before all allowed me to demonstrate my learnings as much as possible while extending my knowledge.
 Thus, I consider this practical a success.