Operating Systems Project 1

B06902135 資工三 蔡宜倫

Introduction

- This project is about scheduling policies: FIFO, RR, SJF and PSJF, I will show my program design, kernel version and compare the reality result with theoretical result below.
- If you find my project a little similar to https://github.com/fslo1709/Project1_OS, that is because I withdrawn the class after I finished project 1 last year due to personal issues.

1. Program Design

- To run the code, please be sure that you are in **root mode** because else you won't be able to print string to dmesg. This is really important, the reason this happened is in **Kernel Version**.
- After scanning the input, I set all process to not active and start checking for ready time of each process in every time unit.
- If the process is ready, fork a child process and get the current time. Also, set scheduler to **idle** in order to block the process and prevent multiple process executing at the same time.
- Depending on the policy, choose the next ready process and switch to it every time a previous process end or try preemptive.
- After each process finishes, get the current time of day. Print the process name and pid to stdout and print the pid and execution time period to dmesg.
- Exit the program after all processes are finished.

2. Kernel Version

- **Linux-4.14.25** (https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.25.tar.xz)
- I wrote a system call to get time of day and tried to pass it to buffer print and print the result in dmesg. While I successfully implemented syscall $Get\_Time.c$, my $buffer\_printk.c$ just can't work, So I came up with another solution: that is to file open the dmesg and print the result as root, this worked as a charm.

3. Reality v.s. Theoretical

- For example, in FIFO_1.txt:

```
FIFO
5
P1 0 500
P2 0 500
P3 0 500
P4 0 500
P5 0 500
```

```
[Project1] 7884 1588081929.119627445 1588081930.379588511
[Project1] 7885 1588081929.119703102 1588081931.602574521
[Project1] 7886 1588081929.119773533 1588081932.814927080
[Project1] 7887 1588081929.126538740 1588081934.16249123
[Project1] 7888 1588081929.136834107 1588081935.207742144
```

  - The first process should run for 1 sec while the reality time is 1.25 sec.
  - It is easy to see that the first process has more latency than the other processes while they should have the same time theoretically. I think it is because that there are preprocess needed to be done before the first process start but the timer was already clicking.
  - The process running order is correct.

- For FIFO_2.txt

```
FIFO
4
P1 0 80000
P2 100 5000
P3 200 1000
P4 300 1000
```

```
[Project1] 7897 1588081935.237252531 1588082130.792980278
[Project1] 7898 1588081935.494535242 1588082142.427374278
[Project1] 7899 1588081935.734533235 1588082144.501788722
[Project1] 7900 1588081935.978535138 1588082146.70790467
```

  - Theoretically speaking, the first process should run for **160 seconds**, and also because of preprocessing the process should execute a little longer than 160 secs. My result is **195 seconds** which is much longer than theoretical time. I think the reason is because while process 1 executes, the execution time is so long that it also spends **more times** on checking if the next process is ready, therefor the estimation error will also be large.
  - The process running order is correct.

- Through the experiment I found that my program always executes longer than the theoretical time, and my analyzation is:
    - Dynamically adjusting the priority of each processes, executing new processes and preprocessing all takes more time.
    - While executing 1000000000 for loops the CPU won't use the exact same time because each process doesn't have the same unit time on each **for loop**, and also, the timer on scheduler won't perfectly sync with process which leads to latency.
    - Since the computer might be executing other process generated by other programs, there is a possibility that our process was halted by context switch while the timer won't extract the time, hence forcing execution time increase.