Operating System - Project 1

By: Wu-Jun Pei (B06902029)

1. Environment

- VM Environment: VirtualBox
  - CPU: 2
  - RAM: 4GB
- OS: Ubuntu 16.04
- Kernel Version: Linux-4.14.25

2. Design

Scheduler

整體架構

我將四個 policy 分開寫成四個 function。雖然寫成四個分開 function，但有依循一定的流程，如下：

```
while True:
    1. 將所有 ready 的 process 加進 queue / priority queue
        - 在此時 create process 並將 block 他 (實作再 process.c 中)
    2. 若正在跑的 process 需要被暫停 (preemptive)，則將他暫停。
        - 使用 blockProcess 的 function
    3. 若現在沒有 process 正在執行，則從 queue / priority queue 取出一個 process 執行
        - 使用 wakeProcess 的 function
    4. 執行一個 unit time
    5. 檢查當前 process 是否完成，若完成則將 process
        - 使用 waitpid 回收 pid
    6. 檢查是否所有 process 都執行完成，若是則結束
```

實作細節

- Utility
  - 實作 `UNIT` 、 `ELAPSE` 兩個 function 計算時間。
  - 實作一個 **Queue** 並提供 `push` 、 `pop` 、 `front` 的 method
  - 實作一個 **Priority Queue** 並提供 `push` 、 `pop` 、 `top` 的 method
- Process
  - Process 的 structure

    ```
    struct Process {
        char name[40];
        int readyTime, executionTime;
        pid_t pid;
    };
    ```

  - 所有與 process 相關的 function，如 `assignCPU` 、 `blockProcess` 、 `wakeProcess` 、 `startProcess` 。
  - `assignCPU` ：將 parent process 指派到 CPU0，並將 child process 指派到 CPU1
  - block / wake 的方式是使用 `sched_setscheduler` 的方式，設定 `SCHED_FIFO` 的 policy 並設定 1 / 99 的 priority。
- Main
  - 排序所有的 process，按照以下方式：
    - ReadyTime，由小到大
      - ProcessName，字典序由小到大
  - 根據不同的 scheduling policy，使用不同的 policy function
- FIFO
  - 使用一個 Queue 維護所有 ready processes
  - 每跑完一個 Process 才找下一個 process 執行
- RR
  - 使用一個 Queue 維護所有 ready processes
  - 設定一個變數 `TTL` 紀錄當下在跑的 process 還有多少時間會被切換。當 `TTL` 為零時，將 process 暫停並重新加入 ready queue，再從 ready queue 找下一個 process。
- SJF
  - 使用一個 Priority Queue 維護所有 ready processes
  - 每跑完一個 Process，找一個 execution time 最小的 process 執行

- PSJF
  - 使用一個 Priority Queue 維護所有 ready processes
  - 每個 unit time 結束後，檢查是否有 execution time 更小的 process。若有，切換到該 process 執行。

System Call

- `kernel/Makefile` ：加上

```
obj-y += my_printk.o
obj-y += my_gettime.o
```

- `include/linux/syscall_64.h` ：加上以下兩行

```
asmlinkage void sys_my_printk(char *s);
asmlinkage void sys_my_gettime(struct timespec *ts);
```

- `arch/x86/entry/syscalls/syscall_64.tbl` ：加上

```
333     common  my_printk               sys_my_printk
334     common  my_gettime              sys_my_gettime
```

- `kernel/my_printk.c`

```
#include <linux/kernel.h>
#include <linux/linkage.h>

asmlinkage void sys_my_printk(char *s){
    printk(s);
    return;
}
```

- `kernel/my_gettime.c`

```
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/timer.h>

asmlinkage void sys_my_gettime(struct timespec *ts){
    getnstimeofday(&ts);
    return;
}
```

3. Result

Theoretical Outcome

我使用一個 python3 的程式計算一個 input 的 theoretical output，結果以 unit time 為單位。

```
python3 evaluation/theoretical.py
```

Error Calculation

首先，先將執行結果轉換成以 unit time 為單位，並使用以下兩種方式定義一個 Process 的執行誤差。

- Timestamp Error
  - Absolute Error (單位為 unit time)

$$\begin{aligned}\text{Absolute Error} = &|\text{Theo. Start Time} - \text{Real Start Time}| + \\ &|\text{Theo. End Time} - \text{Real End Time}|\end{aligned}$$

  - Relative Error (單位為百分比 %)

$$\text{Relative Error} = \frac{\text{Absolute Error}}{\text{Theo. End Time}} \times 100\%$$

- Turnaround Error
  - Absolute Error (單位為 unit time)

$$\begin{aligned}\text{Absolute Error} = &|(\text{Theo. End Time} - \text{Theo. Start Time}) - \\ &(\text{Real End Time} - \text{Real Start Time})|\end{aligned}$$

  - Relative Error (單位為百分比 %)

$$\text{Relative Error} = \frac{\text{Absolute Error}}{\text{Theo. Turnaround}} \times 100\%$$

實際結果比較

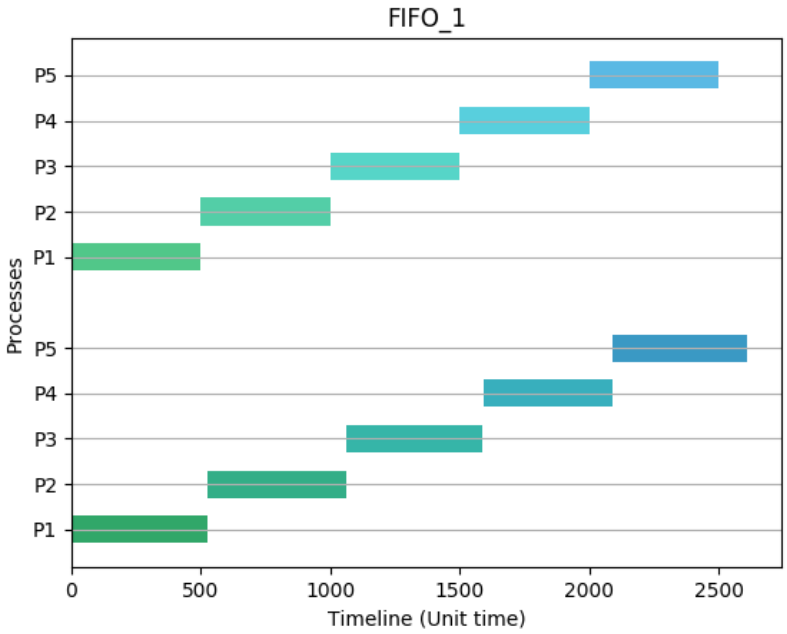
以下展示四個 demo 的 testcase 的結果：

FIFO_1

Proess 第一次被 CPU 排程到的時間、結束時間表格

Name	Theo. Start Time	Theo. End Time	Real Start Time	Real End Time
P1	0.00000	500.00000	0.00000	526.31091
P2	500.00000	1000.00000	527.42677	1058.60335
P3	1000.00000	1500.00000	1060.32034	1588.90728
P4	1500.00000	2000.00000	1589.15729	2088.42533
P5	2000.00000	2500.00000	2088.64196	2610.68563

作圖（表格上半部為理論時間；下半部為實際時間）



結果說明

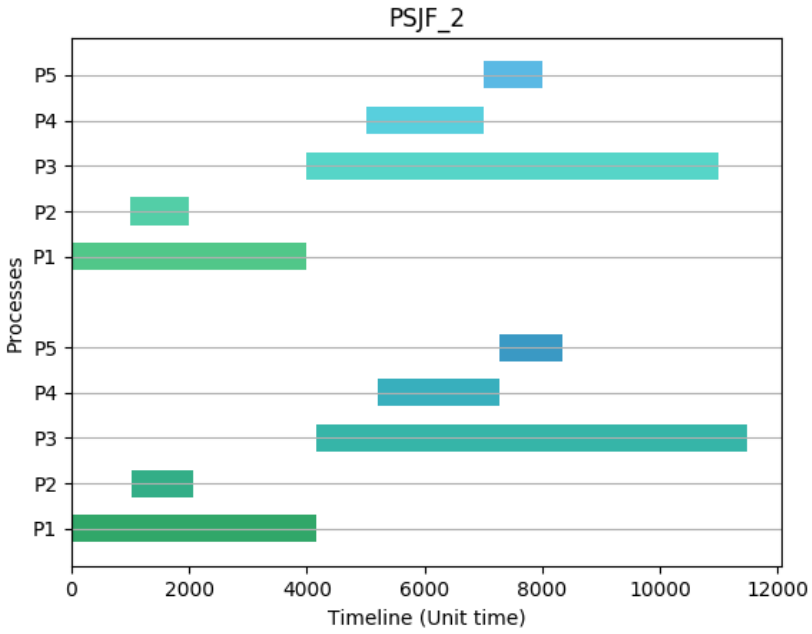
1. 所有的 Process 執行順序、開始時間、結束時間都符合理論值
2. 實際執行起來略比理論值慢一點點
3. 誤差
    - Relative Timestamp Error: 8.13319 %
    - Relative Turnaround Error: 4.35400 %

PSJF_2

Proess 第一次被 CPU 排程到的時間、結束時間表格

Name	Theo. Start Time	Theo. End Time	Real Start Time	Real End Time
P1	0.00000	4000.00000	0.00000	4157.88830
P2	1000.00000	2000.00000	1023.08203	2058.34835
P3	4000.00000	11000.00000	4158.74988	11487.01499
P4	5000.00000	7000.00000	5200.76232	7279.89572
P5	7000.00000	8000.00000	7280.21392	8339.41034

作圖（表格上半部為理論時間；下半部為實際時間）



結果說明

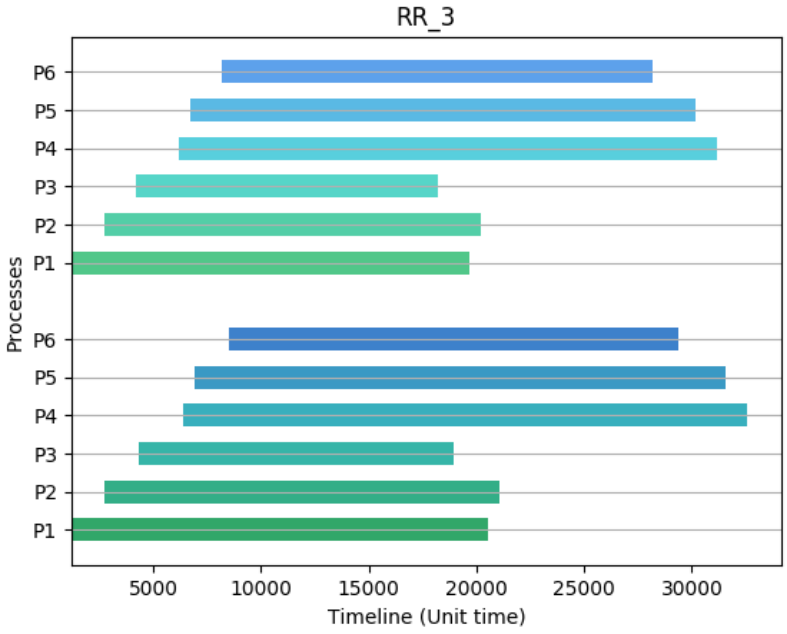
1. 所有的 Process 執行順序、開始時間、結束時間都符合理論值
2. 實際執行起來略比理論值慢一點點
3. 誤差
    - Relative Timestamp Error: 5.70023 %
    - Relative Turnaround Error: 4.40793 %

RR_3

Proess 第一次被 CPU 排程到的時間、結束時間表格

Name	Theo. Start Time	Theo. End Time	Real Start Time	Real End Time
P1	1200.00000	19700.00000	1200.00001	20517.78023
P2	2700.00000	20200.00000	2730.38518	21062.99559
P3	4200.00000	18200.00000	4323.09876	18956.66711
P4	6200.00000	31200.00000	6413.78382	32593.31699
P5	6700.00000	30200.00000	6907.04945	31539.50763
P6	8200.00000	28200.00000	8478.53302	29392.42141

作圖（表格上半部為理論時間；下半部為實際時間）



結果說明

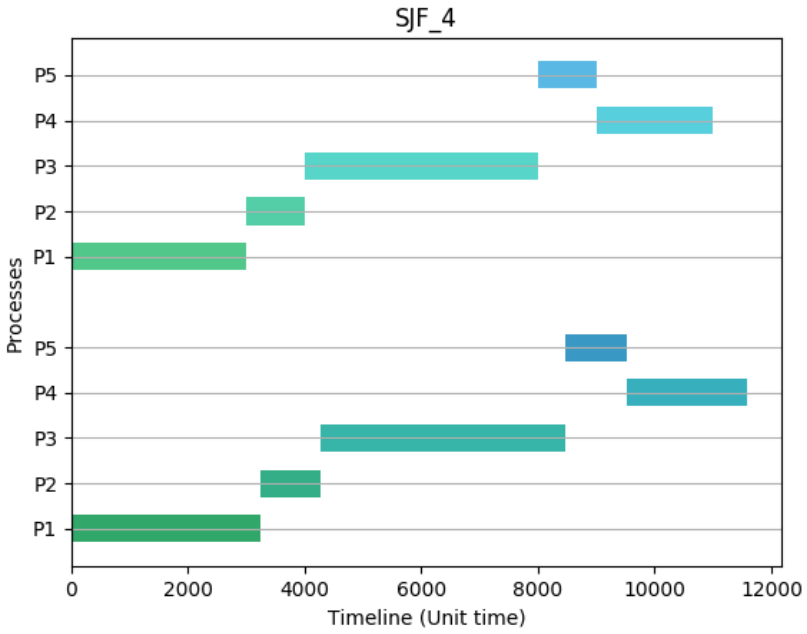
1. 所有的 Process 執行順序、開始時間、結束時間都符合理論值
2. 實際執行起來略比理論值慢一點點
3. 誤差
    - Relative Timestamp Error: 4.81598 %
    - Relative Turnaround Error: 4.63504 %

SJF_4

Proess 第一次被 CPU 排程到的時間、結束時間表格

Name	Theo. Start Time	Theo. End Time	Real Start Time	Real End Time
P1	0.00000	3000.00000	0.00000	3232.67896
P2	3000.00000	4000.00000	3240.60059	4259.95591
P3	4000.00000	8000.00000	4274.54613	8463.12474
P4	9000.00000	11000.00000	9518.56327	11593.14746
P5	8000.00000	9000.00000	8463.43757	9518.33241

作圖（表格上半部為理論時間；下半部為實際時間）



結果說明

1. 所有的 Process 執行順序、開始時間、結束時間都符合理論值
2. 實際執行起來略比理論值慢一點點
3. 誤差
    - Relative Timestamp Error: 10.10116 %
    - Relative Turnaround Error: 4.72493 %

Full Error Table

Testcase	Timestamp Abs. Error	Timestamp Rel. Error (%)	Turnaround Abs. Error	Turnaround Rel. Error (%)
FIFO_1	127.69577	8.13319	21.77001	4.35400
FIFO_2	8143.16738	9.56557	1191.43638	4.45349
FIFO_3	1275.73914	7.54962	141.83908	4.00796
FIFO_4	173.28525	6.43510	31.92426	3.69276
FIFO_5	1301.64695	7.66673	157.60649	4.71152
RR_1	135.32958	8.51449	24.98108	4.99622
RR_2	338.29045	3.81558	338.29044	4.21711
RR_3	1202.58987	4.81598	918.30645	4.63504
RR_4	612.59579	4.69308	509.60035	4.39405
RR_5	730.32608	5.44707	603.91997	4.83008
SJF_1	457.80325	6.80839	163.60156	5.06516
SJF_2	332.43569	3.85704	140.29207	3.51088
SJF_3	902.68048	6.90968	175.00788	3.43338
SJF_4	712.87741	10.10116	114.01838	4.72493
SJF_5	194.91276	6.73422	40.01271	5.39100
PSJF_1	693.88752	5.11107	600.65538	4.85527
PSJF_2	397.07317	5.70023	131.94991	4.40793
PSJF_3	121.29800	6.13651	64.99896	5.99323
PSJF_4	409.12595	5.92006	168.94449	4.56290
PSJF_5	361.05304	3.67889	157.68587	3.26790
Overall	**931.19068**	**6.37968**	**284.84209**	**4.47524**

Conclusion & Discussion

1. 此次 Project 實作的 scheduler 基本上是符合理論上的順序及執行時間的
2. 做出來的結果中可以發現大部分的 process 都比較晚開始，且比較晚結束。我的推測是因為使用 `TIME_MEASUREMENT` 所計算的 UNIT TIME 是比較快的（可能因為排程的 testcase 比較簡單），而到比較複雜一點的 scheduling policy 因為判斷條件較多，而會有誤差。
3. 除了上述問題外，因為我的實作方式使用兩個核心，且我的 VM 只有兩個核心。實驗發現只要有其他 Process 正在執行將造成時間誤差增大，如 `sshd` 、 `tmux` 、 `htop` 等會干擾 schduler、process 的執行，會導致誤差最多到 33%。上面 output 的結果是重新開機後，直接跑以上 testcase 所得到的結果。可以看到有明顯的進步。
4. User-space 的 scheduling 可能無法達成完美零誤差的排程，因為 scheduler 計算 unit time 需要使用 CPU，所以結果會被使用 CPU 的其他 running process 影響。完美的排程方式可能必須寫在 OS 中，並且使用採用其他時間計算方式。