

B07902056 郭瑋喆 資工二

OS_Project1 Report

一、設計

(一) 程式架構

在整體設計上總共有5個檔案，分別是main.c、process.h、process.c、scheduler.h以及scheduler.c。每個部份的分工大致如下：

main.c: 讀入所有資料，以struct process的型式存取，並呼叫

```
int scheduling(process* proc, int count, char* policy)
```

process.h/process.c:

定義 UNIT_T() 以及 struct process，並有4個與處理 process 有關的函式，如下：

```
int assign_cpu(int pid, int core);
```

此函式用於指定cpu，讓排程與執行分開。

```
int exec(process proc);
```

此函式用於 process 的執行，當一個 process ready 的時候，便會由此函式進行 fork() 並放入不同 cpu 執行。process 由此取得 pid（最一開始預設為-1）。

```
int block(int pid);
```

```
int wakeup(int pid);
```

此兩個函式用於調整 process 的優先權。

scheduler.h/scheduler.c:

此部份負責 process 的排程。有兩個函式如下：

```
int next_process(process* proc, int count, char* policy);
```

此函式決定下一個單位時間應執行哪一個 process。

```
int scheduling(process* proc, int count, char* policy);
```

此函式在 main() 讀完資料後呼叫，藉由無窮迴圈來安排 process。

(二) Scheduling Policies

對於四種不同的 policy，在 next_process 中執行不同的策略。分述如下：

FIFO: 愈早 ready 的 process 就愈先執行。由於在 scheduling 的最一開始就先將所有的 process 排序，因此只要從被排在較前面的開始做即可。因此只要當前 running process 尚未結束，就將 next 設為它即可。

SJF: 當工作完成時，挑選其餘中最快的先做。若當前 running process 尚未結束，就和 FIFO 相同，將 next 設為它；若結束，則從已經 ready 的 process 中選出執行時間最短的，將其執行到結束為止。

PSJF: 和 SJF 的差別在於，每當有新的 process ready，便要檢查當前能最快完成的是誰。因此每次決定 next 都要確認是否有恰好 ready 的 process。若當前工作完成，則選擇已經 ready 的工作中最快完成的執行。

RR: 每個 process 每次只能跑一定時間（此為500單位時間），若時間到仍尚未結束就要換下一個 process 先執行。而決定下一個是誰的方法，我並沒有選擇使用 queue 等資料結構來實作，而是在當前 running process 結束（無論是真的結束或者時間到），遍歷所有的 process，選擇已經 ready 的之中等最久的。而紀錄一個 process 等了多久的方法，我是藉由修改 ready_time 來完成：當一個 process ready 但還沒有被執行過，則他的 ready_time 是最一開始得到的；當一個 process 執行時間到達上限，必須換到下一個時，就把他的 ready_time 設為 current_time。接著再從已經 ready 的 process 中選擇等最久的，也就是 (current_time - ready_time) 最大的，因為無論是還沒跑過或者跑到一半，這個值都是該 process 等待的時間。由於 current_time 對所有 process 都相同，因此就是選擇 ready_time 最小的。反覆直到所有 process 均完成。

二、核心版本

linux-4.14.25

三、理論與實際結果比較

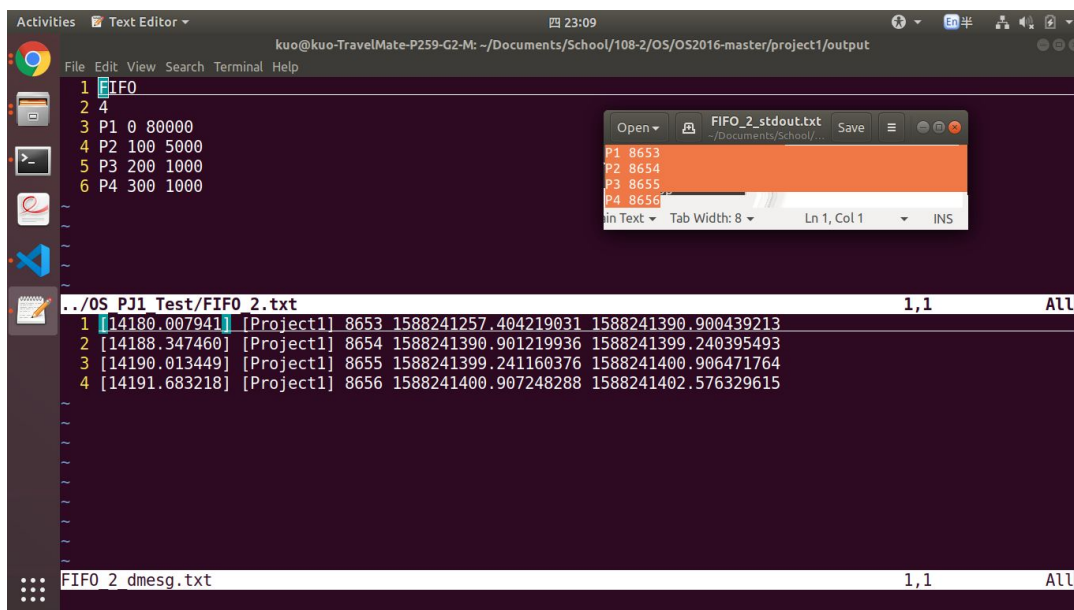
首先，藉由 TIME_MEASUREMENT.txt 這個檔案來計算 UNIT_T()，結果如下。

[illegible]

計算後得到每個 process 跑 500 單位時間平均約花了 0.83884 秒，意即每單位時間大約花費 $0.00167 \approx 1/600$ 秒。

接著挑選一些測資，將理論值與實際值進行比較。

FIFO_2.txt



```
1 FIFO
2 4
3 P1 0 80000
4 P2 100 5000
5 P3 200 1000
6 P4 300 1000

../OS_PJ1_Test/FIFO_2.txt 1,1 All
1 [14180.007941] [Project1] 8653 1588241257.404219031 1588241390.900439213
2 [14188.347460] [Project1] 8654 1588241390.901219936 1588241399.240395493
3 [14190.013449] [Project1] 8655 1588241399.241160376 1588241400.906471764
4 [14191.683218] [Project1] 8656 1588241400.907248288 1588241402.576329615

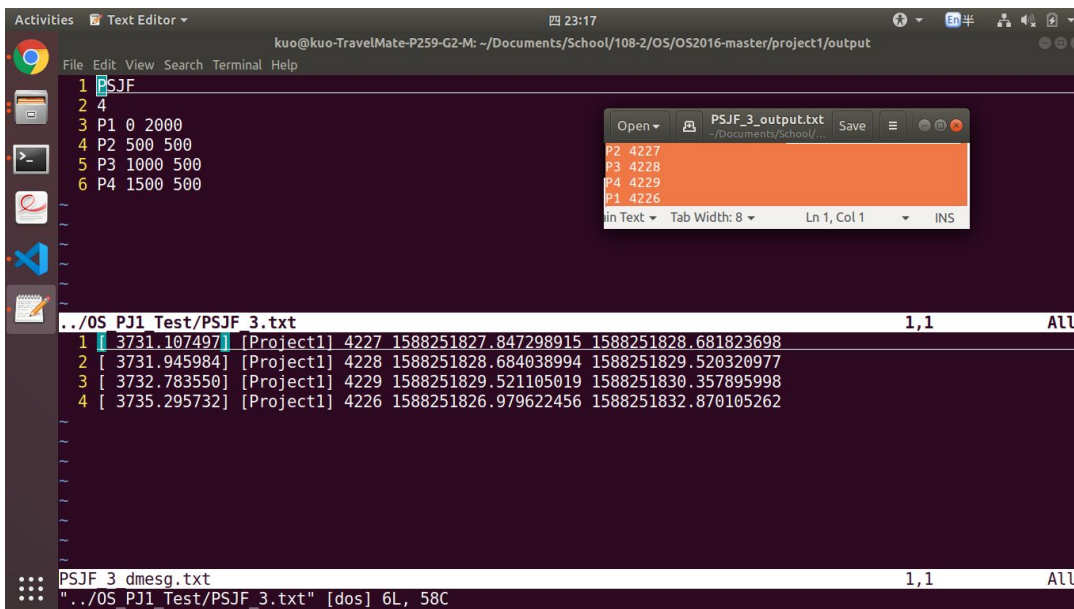
FIFO_2 dmesg.txt 1,1 All
```

完成的順序依序為 p1, p2, p3, p4，與理論符合。

理論上花費的時間依序約為 133.33, 8.33, 1.67, 1.67（秒），而

實際上花費的時間依序約為 133.49, 8.33, 1.67, 1.67（秒），吻合性相當高。

PSJF_3.txt



```
1 PSJF
2 4
3 P1 0 2000
4 P2 500 500
5 P3 1000 500
6 P4 1500 500

../OS_PJ1_Test/PSJF_3.txt 1,1 All
1 [ 3731.107497] [Project1] 4227 1588251827.847298915 1588251828.681823698
2 [ 3731.945984] [Project1] 4228 1588251828.684038994 1588251829.520320977
3 [ 3732.783550] [Project1] 4229 1588251829.521105019 1588251830.357895998
4 [ 3735.295732] [Project1] 4226 1588251826.979622456 1588251832.870105262

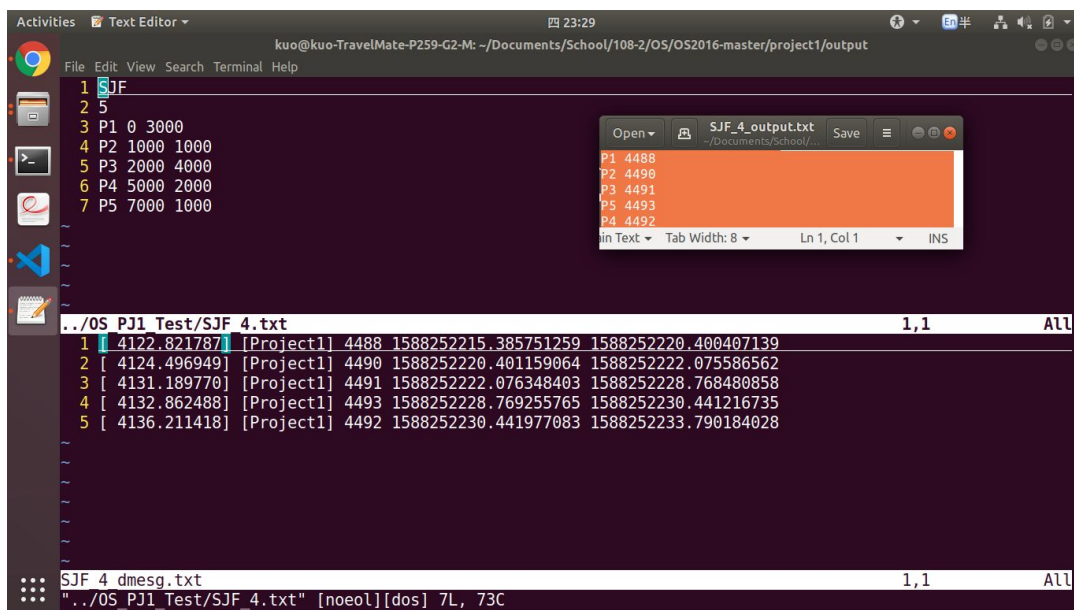
PSJF_3 dmesg.txt 1,1 All
"../OS_PJ1_Test/PSJF_3.txt" [dos] 6L, 58C
```

理論上應該會先執行 p1 500 UNIT_T，接著換到 p2 500 UNIT_T (finish)，接著是 p3 500 UNIT_T (finish)，接著是 p4 500 UNIT_T (finish)，最後回到 p1 完成剩下的 1500 UNIT_T (finish)。執行結果結束的順序為 p2, p3, p4, p1，與理論符合。

理論上花費的時間依序約為 (500+500+500+500+1500) UNIT_T = 5.83, 0.83, 0.83,

0.83（秒），而實際上花費的時間依序約為 5.89, 0.834, 0.836, 0.836（秒），和理論值相差不多。

SJF_4.txt



```
1 SJF
2 5
3 P1 0 3000
4 P2 1000 1000
5 P3 2000 4000
6 P4 5000 2000
7 P5 7000 1000

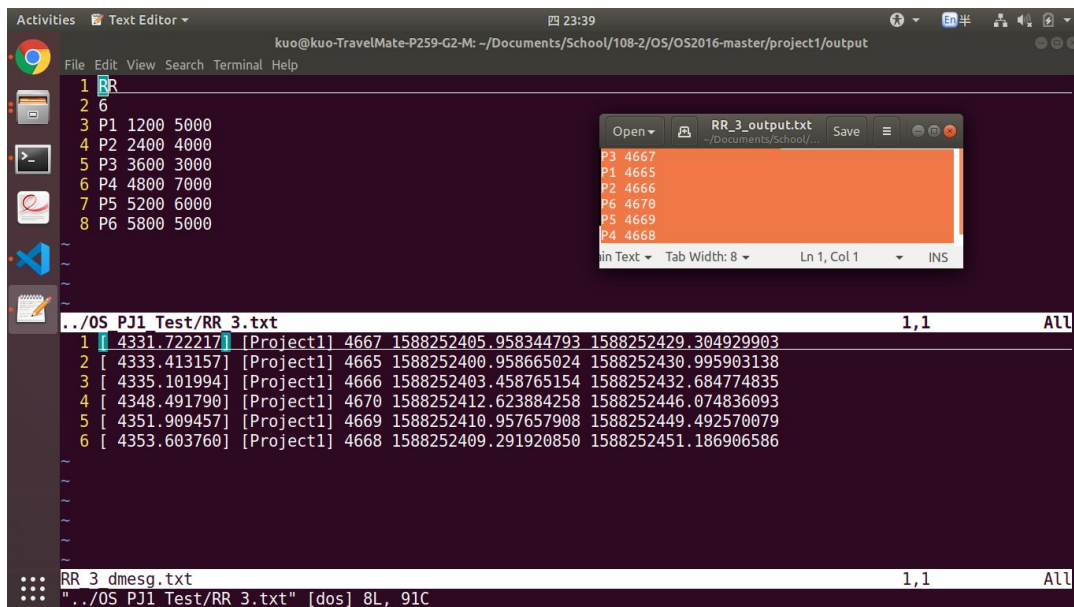
../OS_PJ1_Test/SJF_4.txt 1,1 All
1 [ 4122.821787] [Project1] 4488 1588252215.385751259 1588252220.400407139
2 [ 4124.496949] [Project1] 4490 1588252220.401159064 1588252222.075586562
3 [ 4131.189770] [Project1] 4491 1588252222.076348403 1588252228.768480858
4 [ 4132.862488] [Project1] 4493 1588252228.769255765 1588252230.441216735
5 [ 4136.211418] [Project1] 4492 1588252230.441977083 1588252233.790184028

SJF_4 dmesg.txt 1,1 All
"../OS_PJ1_Test/SJF_4.txt" [noeol][dos] 7L, 73C
```

```
Open SJF_4_output.txt Save
P1 4488
P2 4490
P3 4491
P5 4493
P4 4492
Ln 1, Col 1 INS
```

理論上應該會先把 p1 跑完，接著在 ready 的裡面選出最快的 p2，接著只有 p3 ready 故執行 p3，接著兩個都 ready 了，選擇 p5，最後是 p4。實際結果與理論相符。理論上花費的時間依序約為 5, $(4000-3000)/600 = 1.67$, $(8000-4000)/600 = 6.67$, $(11000-9000)/600 = 3.33$, $(9000-8000)/600 = 1.67$ （秒），而實際上花費的時間依序約為 5.014, 1.674, 6.69, 3.35, 1.67（秒），和理論值差不多。

RR_3.txt



```
1 RR
2 6
3 P1 1200 5000
4 P2 2400 4000
5 P3 3600 3000
6 P4 4800 7000
7 P5 5200 6000
8 P6 5800 5000

../OS_PJ1_Test/RR_3.txt 1,1 All
1 [ 4331.722217] [Project1] 4667 1588252405.958344793 1588252429.304929903
2 [ 4333.413157] [Project1] 4665 1588252400.958665024 1588252430.995903138
3 [ 4335.101994] [Project1] 4666 1588252403.458765154 1588252432.684774835
4 [ 4348.491790] [Project1] 4670 1588252412.623884258 1588252446.074836093
5 [ 4351.909457] [Project1] 4669 1588252410.957657908 1588252449.492570079
6 [ 4353.603760] [Project1] 4668 1588252409.291920850 1588252451.186906586

RR_3 dmesg.txt 1,1 All
"../OS_PJ1_Test/RR_3.txt" [dos] 8L, 91C
```

```
Open RR_3_output.txt Save
P3 4667
P1 4665
P2 4666
P6 4670
P5 4669
P4 4668
Ln 1, Col 1 INS
```

理論上執行的順序應為

111212312341234512345612345612345612456456456456456456456454。其中每個都花費 500 UNIT_T。執行結果的結束順序為 p3, p1, p2, p6, p5, p4，與理論相符。理論上花費的時間依序約為 $35*500/600 = 29.16$, $33*500/600 = 27.5$, $25*500/600 = 20.83$, $50*500/600 = 41.67$, $44*500/600 = 36.67$, $36*500/600 = 30$ （秒）。而

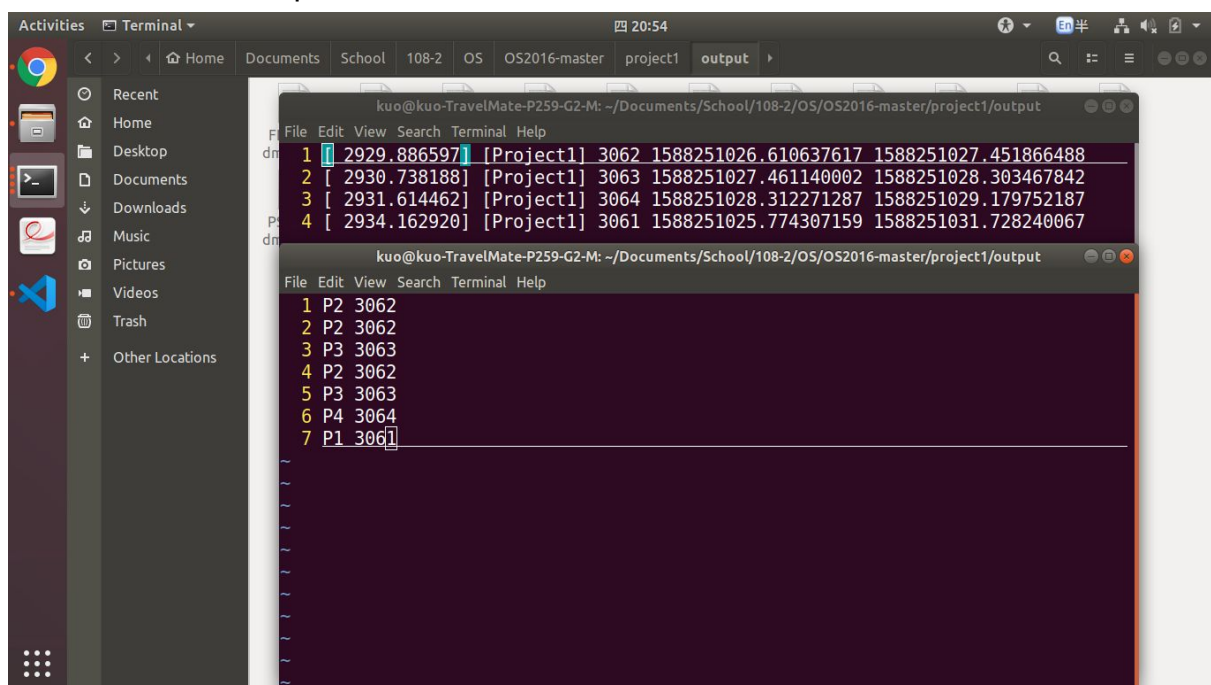
實際上花費的時間依序約為 30.04, 29.23, 23.35, 41.89, 38.53, 33.45（秒），與前面幾個 policy 的結果相比，明顯誤差較大，推測可能是因為在 process 之間做 switch 的次數相對之下多很多，而此時間差累積起來造成了這樣的結果。

四、後記

在寫此作業的過程中，我遇到了一些比較奇怪的問題。

在編譯好 kernel、進入此版本的 OS 執行程式後，卻發現 dmesg 沒有被寫入任何內容。最一開始以為是 syscall 沒寫好，因此寫了一些小程式來進行測試，卻發現並沒有問題。然後經多種測試後，發現問題出在 `exec(process proc)` 中的 `fork()` 有問題，完全不會進到 child process。一開始我懷疑是 code 可能有哪裡發生沒注意到的 bug，然而即使將 `if(pid == 0)` 裡面的內容全部註解掉，只留下測試用的 `printf("child\n")` 和結束 process 的 `exit(0)`，仍然無法進到 child process。反覆不斷檢查整份 code，卻遲遲找不出原因。結果後來隨手開啟 root 模式後，居然就可以了。（不過後來關掉之後又跑不了，花了我不少時間）到目前我仍然想不到問題所在。

後來在輸出 output 檔的時候，發生了奇怪的事情，如下圖：



```
kuo@kuo-TravelMate-P259-G2-M: ~/Documents/School/108-2/OS/OS2016-master/project1/output
1 [ 2929.886597] [Project1] 3062 1588251026.610637617 1588251027.451866488
2 [ 2930.738188] [Project1] 3063 1588251027.461140002 1588251028.303467842
3 [ 2931.614462] [Project1] 3064 1588251028.312271287 1588251029.179752187
4 [ 2934.162920] [Project1] 3061 1588251025.774307159 1588251031.728240067

kuo@kuo-TravelMate-P259-G2-M: ~/Documents/School/108-2/OS/OS2016-master/project1/output
1 P2 3062
2 P2 3062
3 P3 3063
4 P2 3062
5 P3 3063
6 P4 3064
7 P1 3061
```

雖然如果輸出到 stdout 的話，都可以得到正常的輸出內容，但某些檔案卻沒辦法用命令列指令 `./main < XXX_X.txt > XXX_X_stdout.txt` 輸出正確的結果，然而 dmesg 卻是正確的。為此，我重新編譯了 kernel、多次重開機，卻仍然沒法解決，最後逼不得已，只好先輸出到 stdout 再複製貼上到檔案。