# Report

R08922160
山下夏輝

# 1. 設計

## time-driven

之前的作業 homework2 是需要用 event-driven 。但，這個 project1 是用 time-driven 的方式來設計。理由如下。

- homework2 是設定知道 deadline（period）的狀況，因此能夠可以用 event-driven的方式來設計，可以 skip 掉實際上要 execute 的部分，這樣才能比較快速的安排。
- project1 是設定1個 time unit（= volatile unsigned long i; for(i=0;i<1000000UL;i++); ），也需要實際上要跑 execution time 乘以 time unit 的時間，因此需使用 time-driven，反而 skip 掉實際 execution 的 event-driven 是對這個project1老說不適合使用。
- 使用條件分歧並且使用 sched_setscheduler 來控制 priority 能夠讓 priority 高的 task 有限跑起來（有些狀況下讓他跑完）

## system-call my_printk

設計了一個 my_printk，加上 ketnel 裡面。 設計的理由如下。

- 按照 project1 的要求的話，不太能只用 system-call printk 來滿足條件。
- 使用 my_printk 可以 output 出的資訊是這些：TAG PID ST FT。

## system-call gettimeofday

system-call gettimeofday 是取得現在的時間點，因此 task 開始跑的時候取得 start time， task 結束的時候取得 end time。project1 的 specification 裡面是寫 “getnstimeofday”，但是我在 kernel 裡面找不到這個 system-call，在 google 上也沒有查到。但，system-call gettimeofday 本身就能夠取得得到滿足 project1 的條件的程度的時間點。

## mask-cpu

1個 CPU 是程式跑起來一開始就 mask 起來，專門安排 task 的 process 來使用，另外1個 CPU是 child process 拿到 pid 的時候（本來是設定-1）mask 起來，不讓其他 process interrupts child process 正在

跑的 CPU 裡影響到 execution time。

### task scheduling

- 先按 ready-time 排序，如果 ready-time 是一樣的話，按 task-ID 比較小的比較優先排序。
- time-driven 的方式來安排schedule。
    - 每個 time-unit 檢查有沒有 task 結束、如果 preemptive 的的話，檢查要不要 context switch。
    - 如果有再跑的 task 的話，也是每個 time-unit，減它的 execution time。
- 按 policy 先設計條件，後續是基本上現在有沒有再跑的 task、現在有沒有拿著 pid、已經有沒有跑完、各種 policy 的特質來設置條件分歧。

## 2. 核心版本

### HOST

OS: Windows 10 Enterprise
Processor: Intel(R) Core(M) i5-8500 CPU
# of Processor: 6
RAM: 16.0GB

### GUEST (Virtual Box)

OS: Ubuntu(64 bit) with Linux 4.14.25
# of Processor: 4
Main Memory: 2048MB

## 3. 比較實際結果與理論結果，並解釋造成差異的原因

### TIME_MEASUREMENT.txt

Schedule-policy 是 First In First Out，execution-time 全都是 500 time-units，從 0 開始每個 task 的 ready-time 是第 n * 1000 time-units的時候。

> FIFO
> 10

> P0 0 500
> P1 1000 500
> P2 2000 500
> P3 3000 500
> P4 4000 500
> P5 5000 500
> P6 6000 500
> P7 7000 500
> P8 8000 500
> P9 9000 500

# 理論結果

理論上

- 每個 task 的 execution-time 會是一模一樣
- 每個 task 之間的 沒有任何 task（child process）跑在 CPU 上的時間（之後 rest-time）也會是一模一樣
- 每個 task 的 execution-time 和 每個 task 之間的 rest-time 也會是一模一樣

# 實際結果

以下表是在程式上把 TIME_MEASUREMENT.txt 裡面的 tasks 跑的結果。start_time 是 task 開始跑的時間， end_time 是 task 跑完的時間，execution_time 是 end_time 減 start_time 的數值，rest_time 是每個 task 之間的 沒有任何 task（child process）跑在 CPU 上的時間、等於 下一個 task 的 start_time 減剛剛跑完的 task 的 end_time 的數值，avrg_exec_time 是1個 unit-time 平均為了 execution 花的時間，avrg_rest_time 是1個 unit-time 平均為了 下一個 task 開始跑之前等的時間，deference 是 avrg_exec_time 和 avrg_rest_time 的差距，整的話avrg_exec_time比較久、負的話avrg_rest_time比較久。

| | start_time | end_time | execution_time<br>Pn.end_time - Pn.start_time | rest_time<br>Pn.start_time - Pn-1.end_time | avrg_exec_time<br>execution_time / 500 | avrg_rest_time<br>execution_time / 500 | defferences<br>avrg_exec - acrg_rest |
|---|---|---|---|---|---|---|---|
| P0 | 320.000580454 | 321.000405674 | 0.999825220 | | 0.001999650 | | |
| P1 | 322.000324186 | 323.000182181 | 0.999857995 | 0.999918512 | 0.001999716 | 0.001999837 | -0.000000121 |
| P2 | 324.000086633 | 324.000939633 | 0.000853000 | 0.999904452 | 0.000001706 | 0.001999809 | -0.001998103 |
| P3 | 325.000839560 | 326.000707320 | 0.999867760 | 0.999899927 | 0.001999736 | 0.001999800 | -0.000000064 |
| P4 | 327.000525090 | 328.000384528 | 0.999859438 | 0.999817770 | 0.001999719 | 0.001999636 | 0.000000083 |
| P5 | 329.000283143 | 330.000159401 | 0.999876258 | 0.999898615 | 0.001999753 | 0.001999797 | -0.000000045 |
| P6 | 331.000051750 | 331.000929931 | 0.000878181 | 0.999892349 | 0.000001756 | 0.001999785 | -0.001998028 |
| P7 | 332.000794370 | 333.000665289 | 0.999870919 | 0.999864439 | 0.001999742 | 0.001999729 | 0.000000013 |
| P8 | 334.000547430 | 335.000428637 | 0.999881207 | 0.999882141 | 0.001999762 | 0.001999764 | -0.000000002 |
| P9 | 336.000312084 | 337.000177017 | 0.999864933 | 0.999883447 | 0.001999730 | 0.001999767 | -0.000000037 |
| SUM | | | 8.000634911 | 8.998961652 | 0.016001270 | 0.017997923 | -0.001996653 |
| AVRG | | | 0.800063491 | 0.999884628 | 0.001600127 | 0.001999769 | -0.000399642 |

用黃色 high-light 的部分有兩個明顯的差距。

1. task 之間的 deference 數值有相當大的差異。理論上，每個 task 的 execution-time、rest-time 會是一模一樣。比如， P1 和 P2 的 deference 的數值差異。
2. deference 的數值和 0 有相當大的差異。理論上，每個 task 的 execution-time 和 每個 task 之間的 rest-time 也會是一模一樣。比如，P2 的 deference 的數值和 0 有差異。

另外，execution_time 的綜合跟 rest_time 的綜合也有差異，是從 P2 和 P6 的 execution_time 來的。這兩個 task 的 execution_time 非常的段。

## 造成差異的原因

1.和 2.的共同的原因，我能夠想到的是以下幾個。

- 有些 if 等條件分歧的部分，CPU 猜錯 if 條件的 True/False 結果，所以 Flash pipeline 的時間影響到了。
- process 跑的過程中，資料放在 cache 裡頭，但某一些原因需要的資料不是在最靠近的 cache 裡，就影響到了。
- 把 CPU mask 起來的時候，要 mask 的那個 CPU 被其他 process（不是這 task 裡面 process 的其中一個）佔住，mask 的時候多花了時間，就影響到了。

2.的原因，我能夠想到的是以下幾個。

- fork（）之後，用 system-call gettimeofday 來取得時間之間，有一段距離，fork（）之後沒有馬上取得 start_time）。
- 程式上，取得 start_time 跟 end_time 的距離 比 取得 end_time 跟 start_time 的時間比較短，rest_time的時間段有比較多的程式碼在裡面。

另外，一個execution_time 的綜合跟 rest_time 的綜合有差異的部分，我能夠想到的是以下幾個。

- load/write memory/cache 的時間影響到了結果。
- 條件分歧的部分，CPU 猜 True/False 猜對的完美。
- 程式本身有瑕疵，就跳過execution。

# 4. 其他

## demo_video

因為攝影之後產生的本來的 mp4 file 太大，上傳不了 github 上，所以我 reduce 了 file size。我 reduce 好了之後，確認畫質太不好會不會影響到 TA 的工作，但我覺得畫質是不會影響到確認影片裡內容的程度，所以我直接上傳了。

1. 如果畫質不好，影響到評分工作的話，請告知，我會用其他方式傳給您。
2. 因為網路上 reduce了 file size 所以 影片右下角有網站的 water-print。如果這樣的影片是不認可的話，請告知，我會用其他方式來傳給您。
3. 因為沒有指定影片的file format，所以我自己選擇了 mp4。如果不方便的話，我來換format也可，重拍也可，所以請告知
   我使用的網站是以下的。
   Clideo ： https://clideo.com/compress-video

## kernel_files

因為 kernel_files directory 之下的 files 放置方式沒有指定，所以我保持路徑放了指定的files。如果評價的時候，會造成您的不方便的話，我會修改，所以請告知。

## Makefile

因為 Makefile 也沒有詳細的指示，所以我寫了只有 compile 的部分，並且compile之後產生“main”，用“./main”來下指令。如果測驗的時候，會造成您的麻煩的話，我會修改，所以請告知。

## 謝謝看完我的報告

看完到此感謝您！ 雖然我的中文、英文都沒有跟大家很好，但我盡量寫別人容易看懂的樣子。如果您有沒看懂的地方的話，請告知。我會其他表達方式來說明。
這堂課好多好多個學生，TA工作應該很辛苦。加上，也看我的報告，再次感謝您。之後的，學期後半也請多多指教！