

# NTU CSIE OS 2020 Project1

---

- Student Name: 林楷恩
- Student ID: B07902075

a user-space scheduler based on the priority-driven scheduler built in Linux kernel for a set of child processes

## Environment

---

- **Kernel Version:** 4.14.25
- **Platform:** Ubuntu 16.04

## Execution & Tools

---

- 將kernel\_files/裏面的檔案放到正確的地方，並重新編譯kernel
- 執行 make
- `sudo ./scheduler < OS_PJ1_Test/${test_name}`
- 一次跑完全部測試的 shell script: `run.sh` (需要 sudo )
- 比較理論與實際結果的 python3.6 script: `compare.py`

## Design

---

### Overall Structure

- 使用2顆 CPU，一顆(0)用來跑 Scheduler，另一顆(1)用來跑 children processes。
- scheduler.c 只負責 read and preprocess inputs, 將執行的 CPU 及 priority 調整好，就按照policy call對應的函數。對於每一種 Policy，其對應的 scheduler function 定義在獨立的檔案，例如 RR\_scheduler 就定義在 RR.c 裏面。
- preprocessing 的部分包含將 processes 按照 ready time 升序排序，這樣能更方便後續處理 processes 的 arrival。如果出現 ready time 相等的情況，則按照在 input 裏的出現順序決定。這樣排序可以幫助 scheduler 更簡單有效率地決定在目前的時間點有哪些程式可以 run。

### Process Control

我寫了一些 functions 來協助我進行 processes 的控制：

- `proc_set_priority` : 利用 `sched_setscheduler` 設定指定的 priority。( `SCHED_FIFO` )
- `proc_set_other` : 利用 `sched_setscheduler` 設定 policy 為 `SCHED_OTHER` (非 real-time process, 所以 priority 較低)

- `proc_set_cpu` : 利用 `sched_setaffinity` 讓指定的 process 跑在指定的 CPU 上
- `proc_start` : fork 出新的 process，爲了避免他在 scheduler 還沒處理前先跑，一 fork 出來就立刻把它的 priority 設成最低。
- `proc_block` : 把一個 process 的 priority 調成最低。
- `proc_wakeup` : 把一個 process 的 priority 調成最高。
- `proc_term` : 把一個已經結束的 process wait 掉。

## FIFO\_scheduler

- 因爲在上面提到的預處理階段，已經把 processes 按照他們的 ready time 排序過了。所以我們可以直接依照他們在 array 裏的順序來處理。
- 維護一個整數 `cur_p`，負責處理「正在 run 的 process」。如果 `cur_p` 指到的 process 是 ready 的狀態，就把它跑起來。如果是在 running 的狀態，就把 `remaining_time` 減一。如果 `remaining_time` 爲0, 就將其終止(wait)，並把 CPU assign 給下一個 process。

## RR\_scheduler

- 我用 Linked List 實作了一個 Queue，當一個 process 進入 ready state，就把他 push 進 queue 裏，如果當下沒有正在跑的 process，則 pop 一個 process 出來跑。
- 維護一個變數 `counter` 來記錄現在這個 process 還剩多少時間可以跑。每次把一個 process 叫醒，就把 `counter` 設爲 time quantum(500)。並在每個 time unit 之後減一。如果減到零了，且那個 process 的 `remaining time > 0`，就把他 push 回 queue 裏，換下一個 process 跑。

## SJF\_scheduler

- 如果當下沒有在跑的 process，就從所有在“ready state”的 processes 中找出 `remaining time` 最低的 process 來跑。
- 每個 time unit 減少一次現在在跑的 process 的 `remaining time`，直到它變爲 0，終止該程式並 release CPU。

## PSJF\_scheduler

- 如果當下沒有在跑的 process，就從所有在“ready state”的 processes 中找出 `remaining time` 最低的 process 來跑。
- 如果當下有在跑的 process 且它的 `remaining time` 大於在目前已經 ready 的 processes 中的最小 `remaining time`，就 preempt 現在這個 process，改成跑 `remaining time` 最小的那個。
- 每個 time unit 減少一次現在在跑的 process 的 `remaining time`，直到它變爲 0，終止該程式並 release CPU。

## 比較實際結果與理論結果

我寫了一個 python script 去計算理論數值和我的 output 的差別，比較結果放在 `theory_vs_real.txt` 這個檔案裏。如果需要執行的話，請確保 python 版本是 3.6 以上，且 input 和 output 都放在對應的資料

夾裏。對於每一個 process，我以百分比表示實際值相較理論值提升或減少了幾%。我將根據這些數值來說明我的觀察：

- 基本上，實際值略多於理論值，但誤差均小於 5%，這代表我的 scheduler 有確實遵循對應的 policy。而略多的部分我認為是因為 scheduler 的時間實際上會過得比較慢，因為 scheduler 除了計時以外，還需要處理排程的事務，因此 scheduler 和 child processes 沒辦法完全 synchronized 是很合理的結果。這使得有一些 CPU 時間因為 scheduler 來不及分配而浪費掉，所以實際的執行時間就會比較長。此外，因為同時間系統裏還有其他程式在跑，而且這些程式無法被 scheduler 控制，多少會佔用一點 CPU 的時間。
- 有極少數情況，實際的時間會「早於」理論的時間，我認為這是因為我的 scheduler 無法在已經把一個 process fork 出來的情況下仍能保證它不會「意外」佔用到 CPU。因為所有程式的 scheduling 實際上仍是由 kernel 控制的。在極端情況下，甚至可能會出現 start time 遠早於理論值的情況，就是因為那個 process 在意外被分配到的 CPU 時間裏 call 了獲取時間的 system call。對於這個情況，我用了2種方式去確保不會發生：
  - 一把 child process fork 出來，就把它的 priority 設得很低，至少要比預設值低，如此以來就可以讓它很不容易被分配到 CPU。這個做法有一個需要注意的點是 sched\_setscheduler 和 nice 預設都是會繼承的，所以爲了不讓它繼承到 scheduler 的 high priority，要加上 SCHED\_RESET\_ON\_FORK 的 flag。
  - 利用一個 priority 比所有在 ready queue 中的 processes 還要高的 process(我稱其爲 dummy)，去擋住那些不應該執行的 process。如此以來，就算 scheduler 在處理 context switch 時會有一點空窗期，在 ready queue 裏的程式也會因為 priority 比 dummy 還要低而不會被分配到 CPU。