B07902014 資工二 蔡承濬

# 設計

以雙核心架構實作，共分成三大部分：

- **main.c**

  讀取 input 並指定 policy 給 scheduler。

- **myfunctions.c**

  提供函式給 scheduler 使用。

- **scheduler.c**

  執行排程。

每個部分的細節如下：

## main

讀入 input 並將排程方法轉換為相對應的編號後傳給 scheduler。

## myfunctions

- **proc_assign_cpu**

  使用 `sched_setaffinity()` 指定特定核心給某一個 process。

- **proc_exec**

  使用 `fork()` 產生已經 ready 的 process。並 block 住子程序它直到被 scheduler 排程到。

  - 子程序（利用 `proc_assign_cpu()` 將子程序指定到專用的 core）：

    被 block 住直到被 scheduler 排程到。在被喚醒時抓取時間，並在跑完指定的 time units 後再次抓取時間，用 `printk()` 印出。

  - 父程序：

    回傳子程序的 pid。

- **proc_activity**

  根據參數決定要調升抑或調降某 process 的優先序，使用 `sched_setscheduler()` 實作。

## scheduler

- **proc_next**

  回傳下一個 time unit 該跑哪一個 process 的 index，四種 policy 的方法如下：

  - **FIFO**：

    屬於 Non-preemptive，故若有 process 正在跑會直接回傳當前 index。否則跑一個迴圈尋找下一個已經 ready 的 process。

  - **RR**：

若沒有 process 正在執行就按順序挑已經 ready 的 process；若目前進行中的程式已經跑了 time quantum 個 time units，就尋找下一個還沒做完且已經 ready 的 process；剩下的情況為程式進行中且還沒跑到時間限制，直接回傳當前 index 即可。

- **SJF**：

  屬於 Non-preemptive，故若有 process 正在跑會直接回傳當前 index。否則尋找已經 ready 且尚未完成的 process 中所需時間最少的回傳。

- **PSJF**：

  尋找已經 ready 且尚未完成的 process 中所需時間最少的回傳。

- **proc_scheduling**

  運用上述所有函式來排程，前置作業包括：

  - 將所有 process 的 pid 設為 -1，代表尚未 ready。
  - 利用 `proc_assign_cpu()` 將 scheduler 指定到其專用的 core。
  - 將 scheduler 優先度調高
  - Initial 紀錄時間與 process 狀態的變數

  接著開始跑一個無窮迴圈，在所有 process 完成後跳出，其內容如下：

  - 檢查當前程式是否完成，若是，呼叫 `waitpid()` 收取其結束資料後，印出它的名字與 pid，並調整相對應的變數。
  - 跑一個迴圈，檢查現在這個時刻有沒有人 ready，並呼叫 `proc_exec()` 執行它之後，將其優先度先降低，等待排程。
  - 呼叫 `proc_next()` 得到下一個應該要跑 process index，若非當前執行中的 process，就進行 context switch，利用 `proc_activity()` 把原本在跑的優先度降低，下一個該跑的優先度提升。
  - 跑一個 time unit。
  - 若有 process 正在執行，則將其還剩下的執行時間 -1。

### Bugs and Resolution

- System call 編不過：

  User space 與 kernel space 的變數間無法直接賦值，故需使用 `copy_to_user()` 將從 kernel 讀出來的時間寫回去。

- 子程序會偷跑：

  一開始測試時，發現子程序會在 `fork()` 後，被降低優先序前，先跑一點點。若發生在所需執行時間長的情況下並無大礙，但若所需時間短，會導致它直接做完，進而使結束順序錯誤。所以，後來使用類似 lock 的概念，子程序會先被 block 在一個 while 迴圈中，若想跳出迴圈繼續執行，則需等待 scheduler 送 signal 來，而 scheduler 只會在該子程序需要執行時呼叫 `kill()`，進而解決偷跑的問題。

## 核心版本

執行 `uname -a` 得到：

Linux ubuntu 4.14.25 #11 SMP Sun Apr 26 04:15:27 PDT 2020 x86_64 x86_64 x86_64 GNU/Linux

# 比較

觀察結果可發現，就算每個 process 都跑同樣的 unit times，實際上的時間也不盡然相同。這可能是因為我們寫的 scheduler 依舊是在 user space 下的操作，真正在執行排程的仍然是 kernel，因此可能在程式運作期間 CPU 仍然被 context switch 到別的 process 我們也無從得知。此外，使用雙核心實踐也可能因執行 context switch 時產生的 overhead 導致 scheduler 和 process 的時間不夠同步，使理論值和實際表現產生偏差，但整體而言誤差並不大，process 結束的順序也和預期相同。