OS Project 1

B07902022 張鈞堯

1. 設計

可以分成三個大函式 `main`, `schedule`, `select_next_process` 以及數個小函式

int main(int argc, char *argv[])

主要將input讀進來，呼叫 `schedule` 函式，並把input傳進去

int schedule(Process *proc, int num_procs, int policy)

先將 `proc` 依照 `ready_time` 排序，並初始化一些變數。因為是兩顆CPU，所以先將現在這個 process assign 到 CPU 0 並提高優先度。接著用一個 `while(1)` 迴圈處理 schedule。

In while loop (終止條件：所有 child process 跑完)

1. 首先先看有沒有 child process 跑完，若有就先 wait 他。
2. 接著看有沒有 process 的 ready_time 等於現在的時間，若有則呼叫 `create_process()`，然後調低它的優先度，如果 policy 是 RR，則將他 push 進 queue 裡。
3. 接著呼叫 `select_next_process()`，得到下一個要跑的 process，如果不是現在正在跑的 process，則進行 context switch，先降低正在跑的 process 的優先度，然後送 signal 給要跑的 process，讓他 unlock，最後提高要跑的 process 的優先度。
4. 呼叫 `unit_time()`，正在跑的 process 的 exec_time--，現在時間 ++。

int create_process(Process pr)

呼叫 `fork()` create child process。

child process

1. 先用一個 lock 讓他 block 在那邊，避免他偷跑。
2. unlock 後，呼叫 syscall(MY_TIME...)，得到 start_time。
3. 用一個 for 迴圈，呼叫 `unit_time()`。
4. 再次呼叫 syscall(MY_TIME...)，得到 end_time。
5. 呼叫 syscall(MY_PRINTK...)，將東西用 `printk()` 印出來。
6. `exit(0)`。

parent process

1. 將child process assign 到 CPU 1。
2. return pid。

int select_next_process(Process *proc, int num_procs, int policy)

1. 如果現在有 process 正在跑，而且是 SJF or FIFO，那就直接 return 正在跑的 process。
2. 如果是 SJF or PSJF，就return ready 且剩餘 exec_time 最少的 process。
3. 如果是 FIFO，就return ready 且 ready_time 最早的 process。
4. 如果是 RR，分成以下情況：
    a. 如果現在沒有 process 在跑，就從 queue 裡 pop 出一個 process。
    b. 如果現在的 process 已經跑 500 unit_time，則從queue 裡 pop 出一個 process，然後把原本在跑的 process push 進 queue 裡。
    c. 上述條件都不成立，則 return 正在跑的 process。

2. 核心版本

4.14.25

3. 比較實際結果與理論結果

首先計算一個 unit_time 的理論值，接著用 unit_time 理論值算出 testcase 的理論值，並與實際值做比對，可以發現實際結果會比理論結果大一點，原因可能是我們的 CPU scheduler 是在 user space 下運行，實際執行 scheduling 的是 kernel，所以可能在 run process 的時候發生 context switch。此外由於我使用雙核心實作，可能會因為 context switch 導致排程的 CPU 跟執行 process 的 CPU 的時間不同步，導致理論與實際的差距。但整體看來，誤差並不大，也不會導致 process 結束的順序與預期不同。