Report

B07902060 趙雋同

設計

- main.c

  讀取測資並且依照process的readyTime作升序排列(假如readyTIme相同則以execTime排序)

- scheduler.c
  - schedule():
    - 將scheduler安排在schedulerCPU裡並且提高priority
    - 執行無窮迴圈直至所有processes全部完成
    - 假若currentTime等於某process的readyTime，便fork出child process
    - 假若nextIndex不等於runningIndex，便對兩個process的priority作調整
    - 假若runningIndex的process執行時間結束，利用signal以及wait處理process，並印出processName和processPID
  - nextProcess():
    - 依照schedulingPolicy的差別分別採用不同的選擇方式
    - RR policy中假如CPU有閒置過，會從readyTime比較小的process開始選擇；否則會以正在執行的process的下一個開始作選擇(即從processIndex+1開始)

- process.c
  - int gotSiganl, allowExecution:
    - gotSignal用來block住process，讓process在被scheduler fork出時不會先進入unitTime的迴圈中偷跑
    - allowExecution用來讓process不會提早死掉，防止不會有閒置的process在scheduler提高他的priority前進入排程
  - assignCPU():

        - 將指定的pid 分配到指定的CPU上

    - setHighPriority(), setLowPriority():
        - 提高以及降低指定pid的優先度

    - initProcess():
        - fork出process並且執行execTime長度的unitTime迴圈
        - 利用GETTIME syscall在第一次進入迴圈時(也就是第一次進入CPU執行時)紀錄起始時間並且在結束迴圈後記錄結束時間，再利用PRINTK syscall輸出至dmesg
        - 等待scheduler傳出的signal後才exit(0)

核心版本

- Kernel : 4.14.25
- OS : Ubuntu 16.04

實際結果與理論結果

依照理論推測的話，scheduler以及各processes每單位unit time在現實中會是同樣長度，但是因為程式碼的instruction不一樣，譬如scheduler會需要判斷下一個process該是哪個，但process只需要跑unitTime迴圈。並且所有processes在childCPU裡必須context switch，而scheduler在parentCPU裡則不須讓出CPU，所以可能會發生schedluer跟process asynchronous

而由於asychronous，有可能導致childprocess在第一次進入unit time迴圈時，當下並不是他優先執行。並且也有可能因為process比scheduler預期的早結束，CPU會直接挑選等待中的processes之一，而不是依照scheduling policy去挑選下一個執行的process，所以我才會利用signal去讓整個scheduling更符合理論。