OS project1 report

設計

main.c

主程式在第0個core，每經過一單位時間，就檢查是否有process開始或結束，然後依據不同的scheduling policy決定下個單位時間應該由哪個process執行。

檢查開始：主程式會記錄經過了幾個單位時間，如果某個process的開始時間等於這個時間，就會新增這個child process，先把它暫停，然後把它的pid和開始時間(**即process的產生時間**)存起來。

檢查結束：每經過一單位時間，就會把這個單位時間執行的process的execution time減1，如果扣到0就代表這個process結束了，此時會waitpid()收回pid。

下個單位時間執行的process：如果下個單位時間的process和上個單位時間的不一樣，就暫停上個process並啟動這個process。而如何選擇這個process如下：

FIFO：如果現在沒有process在執行，則選擇可執行的process中ready time最小的

PSJF：選擇可執行的process中execution time最小的

RR：在這個policy中，把ready time當作進入waiting queue的時間。如果現在時間減ready time=time quantum(500)，則會把這個process的ready time設為現在的時間，然後選擇可執行的process中ready time(即進入queue的時間)最小的

SJF：如果現在沒有process在執行，則選擇可執行的process中execution time最小的

process.c

新增process：fork()一個child process並設在第1個core避免和主程式互相干擾。這個child process執行execution time的單位時間後記錄結束時間並輸出到dmesg。

暫停process：使用SCHED_FIFO並把priority設成1。

啟動process：使用SCHED_FIFO並把priority設成99。

*Note：主程式的priority也是設成SCHED_FIFO的99以確保兩邊的單位時間一樣長。

syscall

我新增了兩個system call：

333：得到現在的時間(second*1000000000+nanosecond)

334：印出規定的資訊到dmesg

核心版本

ubuntu linux-4.14.25

差異比較

以output資料夾中的結果作計算

TIME_MEASUREMENT

一單位時間為0.002574061917秒

以下資料為(轉換後)實際開始-結束時間/理論開始-結束時間

FIFO_1

P1: 0-499.1/0-500

P2: 0.0-1016.9/0-1000

P3: 0.0-1534.5/0-1500

P4: 0.0-2108.1/0-2000

P5: 0.0-2526.0/0-2500

PSJF_2

P1: 0-4091.8/0-4000

P2: 979.4-2005.7/1000-2000

P3: 2005.8-11272.8/2000-11000

P4: 5019.0-7067.6/5000-7000

P5: 7067.7-8099.8/7000-8000

RR_3

P1: 1200-19404.8/1200-19200

P2: 2346.2-20644.4/2400-20200

P3: 3564.4-18105.2/3600-18200

P4: 4694.3-32302.0/4800-31200

P5: 5102.3-31293.3/5200-30200

P6: 5697.9-28916.1/5800-28200

SJF_4

P1: 0-3143.1/0-3000

P2: 906.9/4200.9/1000-4000

P3: 1897.9-8262.9/2000-8000

P4: 5167.1-11406.6/5000-11000

P5: 7239.2-9332.3/7000-9000

由以上的結果可看出實際結果和理論結果大致相符，只是實際的時間稍微長一點。推測是因為選擇下個單位時間要執行的process需要時間，所以沒辦法無縫接軌，而context switch也需要時間，因此造成延遲。Round Robin要切換較多次，所以延遲的時間也比較長。