OS Project 1 report

B07901184 陳映樵

設計：

架構：

1. 我寫了兩個system call: sys_get_time() 和 sys_print_info()。這兩個函數分別會把時間讀進來以及在system上面輸出(dmesg).
2. 我額外實作了一個list以及priority queue去儲存scheduling要用的資訊。不過因為這兩個資料結構都比較基本，所以我沒有額外加一些判斷的東西，所以導致之後程式執行時有的時候結果會怪怪的，雖然不能說是錯的，但是跟預期的不太一樣。
3. 我一開始把資料先讀進我額外寫的一個 tasks_t 結構裡面，順便紀錄是用什麼樣的scheduling algorithm以及task的數量。不過在這個結構裡面，我只是用一般的array去存而已，而要用什麼另外的資料結構就交由scheduler去更進一步處理。
4. 在此project中，我是利用setpriority()來調整所有的priority。

Scheduler：

1. 在不同的演算法中，都有一個scheduler去調整priority以及控制time_unit，不過基本上scheduler只是用來做這些基本的事，至於child process何時結束就是child process自己的事。每一種演算法的差異只是如何調控data structure以及priority而已。
2. 在child process中，child會根據自己的execution time去執行，只有開始以及結束時會呼叫system call去紀錄時間以及在system中寫資訊，其他就沒有什麼額外的事情。
3. 除了child會紀錄時間之外，parent也會紀錄自己的時間，雖然這樣好像會不準，不過在4.我有更進一步的說明。因此parent可以知道什麼時候要create新的child。另外，我把parent放在core1，children都放在core0，因此即便children以及parent的時間不一定完全準確，整體之下還是相當接近的。

4. 由於我們是用priority-driven的方式去做scheduling，那麼當一個process完成時必須要能夠知道以及接著調整其他process的priority。那麼parent是如何知道child結束呢？我在這邊使用waitpid(pid, NULL, WNOHANG)去讓parent與child同時進行，在每次parent執行時，會去看那個child結束了沒有，如果結束了，那麼就挑下一個並把他的priority設成最高，不然的話就等等看有沒有其他是要做。因此我們可以這樣說，parent的timestamp唯一用途便是為了在有process到達時就create他，不然就是是情況調整priority。
5. 因此，由於是priority-driven，當一個process的priority很低時，就幾乎不會動，而且我設計只有一個process會擁有最高的prioriry，所以基本上其他process完全不會做事，直到現在那個process結束為止。
6. 在FIFO以及SJF中，因為不需要考慮Time_Quantum的問題，所以相對簡單，只要把一個process從data structure中拿出來，接著除非要create新的process，不然就試試看那個process執行完沒有。等到整個data structure都空了以後，才會結束整個程式。
7. 在RR以及PSJF中，因為要考慮Time_Quantum的問題，所以除了要看看process有沒有結束，如果當超過一個Time_Quantum時，就要強制調整現在process的priority以及找一個新的process來當新的執行者。

核心版本： 4.14.25

比較：

FIFO :

跟實際比較起來相差不遠，因為實作比較簡單，並且機制也不是很複雜。

RR :

這個版本應該是與實際相差最多的，以RR_1為例。

Input :

RR_1

5

P1 0 500

P2 0 500

P3 0 500

P4 0 500

P5 0 500

Stdout :

P2 26770

P3 26771

P4 26772

P1 26769

P5 26773

即便他們是同時進來的，但是因為可能是compiler優化的原因，導致雖然時間一樣，但是丟進list裡面或是從list裡面獲得priority的順序不同，導致會出現這種次序顛倒的情況發生，不過performance沒有明顯的變化，只是次序變了。不過值得一提的是，有的時候次序就會按照與其的呈現出來，因此這是我懷疑問題在compiler身上的原因。

除此之外，因為要不斷輪換的關係，所以在執行的時候overhead會比較大，因此速度上會稍微比理論上慢一點。

SJF：

在大部分的情況之下都跟預期的一樣，只是問題跟RR一樣，在有多個同樣時間的process之下，執行順序會略微改變，這可能是因為我沒有handle好priority queue的問題，不過我已經沒有心力去做這件事了。雖說如此，但是倒也不會造成什麼困擾，因為只是次序交換，在實際情況下幾乎不會有影響。

PSJF：

跟RR一樣，這種演算法因為要不斷交替以及處理額外的狀況，因此結果比預期的稍微慢一點。

All：

雖然說不同的演算法有不同的瑕疵，但是有一些問題還是每個演算法都會出現的。

最主要的還是parent和child不同步的問題。這個問題導致有的時候會出現一些「空窗期」，也就是說，沒有任何一個process的priority是最高的，大家都一樣，因此在那一小段時間裡，所有的process會輪替著執行。不過因為造成的空窗期很小，因此對peformance的影響不是很大，correctness也只是偶爾會稍微不正確而已。

心得：

我覺得在這個project中最困難的應該就是要搞清楚不同process之間的互動，因為有一度我程式的order不對，導致priority被覆蓋掉，因此讓結果看起來就像一般沒調整過的一樣。這讓我了解到在寫這種程式時，race condition有多嚴重，常常我認為比較簡潔的寫法往往會因為不同的執行順序導致結果不同，因此也學到了之後要如何避免這類型的問題產生。