

Operating Systems Project 1 Report

B06902048 資工三 李峻宇

設計

我的程式首先從 standard input 讀進排程資訊，接著將讀進來的資訊先做處理 (按照 ready time 排序、放進結構等等) 後，呼叫 scheduling() 這個函式進行排程。

我的虛擬機有兩顆 CPU，我讓 scheduling() 跑在 0 號 CPU 上面，而被排程的 processes 都跑在 1 號 CPU 上面。我們利用調整 fork() 出來的 process 的優先度來控制目前換誰跑，也就是說將我們目前計畫使用 CPU 的 process 的優先度調高，其他 process 的優先度則維持是低的，以達到讓被排程到的 process 使用 1 號 CPU。

在 scheduling() 被呼叫之後，會先把自己的 affinity 設定在 0 號 CPU 上，接著呼叫 fork()，創造出 default runner 並且設定 affinity 為 1 號 CPU。增加這個 default runner 用意在於，如果沒有 process 在使用 1 號 CPU 的話，那就是 default runner 在使用，這樣的設計可以幫助我們確保不會有人偷跑。接著 scheduling() 會初始化一些資訊 (剩餘執行時間初始為預計執行時間、time = 0 等等) 並進入一個 while() 迴圈，直到所有該被排程的 processes 都順利結束。在 while() 迴圈裡面，scheduling() 會依序做四件事：

1. 檢查是否有完成的 process。如果有，呼叫 wait() 並且喚醒 default runner，再把已完成的數量加一。這邊我們為了 RR 時的方便，會在計算並記錄下一個預期跑的 process 是誰。
2. 檢查是否有剛抵達的 process，也就是利用判斷 time 是否等於某個 ready time。如果有，呼叫 fork() 並且設定其 affinity 為 1 號 CPU，丟進 ready queue。在第 2 步中我們呼叫 fork() 之後，生出來的 process 會依序做以下事情：
 1. 利用我們自己定義的 system call 來取得系統時間(getnstimeofday())，並且紀錄成被生出來的開始時間。
 2. 將自己的 priority 設成低的，目的是等待別人將自己的 priority 設高之後才可以開始第 3 步。
 3. 根據自身排程資訊執行相對應次數的 time unit。
 4. 利用我們自己定義的 system call 來取得系統時間(getnstimeofday())，紀錄成完成時間。
 5. 將開始時間和完成時間依照指定的格式，透過我們自己定義的 system call 印在 dmesg 上 (printf())。
 6. 呼叫 exit()。
3. 根據 policy 檢查是否需要換別的 process 跑，也就是檢查是否需要 context switch。透過交換 priority 來完成 context switch。這邊須確保同一時間只有一個高優先度的 process，其他都會是低優先度的。根據 policy 的實作如下：
 - FIFO：因為先跑的就可以一直跑到主動放棄 CPU 或是結束，在這邊基本上可以直接 return。若正在跑的是 default runner，我們會從所有 ready queue 選擇最早抵達的 process 來跑。放進 queue 的原則就是先抵達的先放進去。
 - RR：我們在這邊有特別多一個變數來紀錄這個 process 上次被喚醒的時間。若上次被喚醒的時間和目前時間相差是 500，則會喚醒 ready queue 裡的下一位。這邊我們放進 ready queue 的原則是先 ready 的先放，若是因為已經跑了 500 單位時間的 process 我們會直接放進 queue 的會後面。若正在跑的是 default runner，則將我們在 1. 紀錄的 process 喚醒。

- SJF : 因為先跑的就可以一直跑到主動放棄 CPU 或是結束，在這邊基本上可以直接 return。若正在跑的人是 default runner，我們會從 ready queue 裡選擇一個剩餘執行時間最短的 process 來跑。實際上，若是我們的 queue 維持得好(e.g. priority queue)，可以直接選擇從 queue 的最前面拿出來。若正在跑的是 default runner，我們一樣從所有 process 選出剩餘執行時間最短的 process 來跑。
 - PSJF : 因為是 preemptive 的，每次在這裡我們都要檢查剩餘執行時間最短的是哪個 process。實際上，若是在這個時間點沒有新的 process 加入，我們可以相信目前正在跑的 process 還會是剩餘時間最短的。若正在跑的是 default runner，我們就像 SJF 一樣選出一個剩餘執行時間最短的 process 來跑。
4. 跑一個 time unit，並且將 time + 1，這次執行的 process 的剩餘執行時間減一。time 的用途在於紀錄目前的時間，單位為 unit time。

當所有該執行的 process 都完成後 scheduling() 會殺死 (kill) default runner，然後因為檢查條件失敗會跳出 while()，之後就馬上結束。

核心版本

Linux-4.14.25 (<https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.25.tar.xz>)

實際 vs 理論

我們注意到實際跑出來的結果和理論上會有一些差異，這些差異主要反映在時間。完成的先後順序上理論上和實際是相符合的。這邊我們用以下例子來說明。

在 TIME_MEASUREMENT.txt 這份測資中，執行結果如下：

```
1 [Project1] 2183 1586682314.506135542 1586682315.250739706
2 [Project1] 2184 1586682316.040486013 1586682316.824707474
3 [Project1] 2185 1586682317.603940955 1586682318.333880845
4 [Project1] 2186 1586682319.110596108 1586682319.926268945
5 [Project1] 2187 1586682320.687035562 1586682321.435749575
6 [Project1] 2188 1586682322.406568454 1586682323.135658585
7 [Project1] 2189 1586682323.959551140 1586682324.715937854
8 [Project1] 2190 1586682325.441400061 1586682326.417525428
9 [Project1] 2191 1586682327.158623462 1586682327.961150223
10 [Project1] 2192 1586682328.736370063 1586682329.541612456
```

若我們把所有 process 的結束時間減去開始時間，我們會得到如下：

```
1 0.735394
2 0.797179
3 0.729298
4 0.797896
5 0.787567
6 0.733981
7 0.736663
8 0.789898
9 0.733799
10 0.756845
```

一個 time unit 約落在 0.73 ~ 0.8 左右，是個有點大的浮動。當電腦忙的時候，有時甚至可以看到超過 1 的數字。

再來，因為我們的測資是每 1000 單位時間送出一個要跑 500 單位時間的 process，我們可以預期若是把相鄰的 process 的抵達時間相減和完成時間相減的話，會得到結束減去開始的兩倍左右的數字，如下：

```
1 readyTime[i] - readyTime[i - 1] : finishedTime[i] - finishedTime[i - 1], i
  = 1..8
2 (2-1)    1.524375                :          1.586161
3 (3-2)    2.268221                :          2.200339
4 (4-3)    1.515597                :          1.584195
5 (5-4)    1.536851                :          1.526522
6 (6-5)    1.747371                :          1.693785
7 (7-6)    1.509945                :          1.512628
8 (8-7)    1.524678                :          1.577913
9 (9-8)    2.018142                :          1.962043
10 (10-9)   1.514866                :          1.537912
```

可以發現到每個 1000 時間單位真正的時間差都比 500 時間單位的兩倍再多了一點點。

另外，在 RR_3.txt 這份測資中，執行結果如下：

```
1 [Project1] 2326 1586682568.836694051 1586682591.325384366
2 [Project1] 2324 1586682565.233058579 1586682594.668868994
3 [Project1] 2325 1586682567.009590955 1586682595.452022071
4 [Project1] 2329 1586682572.273711239 1586682606.998713040
5 [Project1] 2328 1586682571.355396591 1586682610.078787303
6 [Project1] 2327 1586682570.784389555 1586682611.753150254
```

利用這次 RR 的執行結果算出從開始到結束的時間，和理論上的時間相比(一個 time unit 用 0.75 秒來計算)：

1	name	exepected time(second)	real time(second)
2	P1	28.50	29.44
3	P2	27.45	28.44
4	P3	21.90	22.48
5	P4	39.60	40.97
6	P5	37.50	38.72
7	P6	33.60	34.73

可以發現實際的時間都比預計來的多一些。

因此我們有了以下三點推測，造成理論跟實際的差異：

1. 在我們的設計中，scheduling() 在每一次的 time unit 中會做四件事情，而 fork() 出去的 process 在 time unit 中只單純地跑 for loop 而已，所以會造成 fork 出去的 process 真正跑一次 time unit 跟 scheduling 覺得他跑一次 time unit 會有一些時間上的差異。簡單來說，若 fork() 出去的 process 跑一次 time unit 所需要的時間是 1，那麼 scheduling 所需要的時間大約會多一點，約是

1.02 ~ 1.04。在這樣的差異下，當跑的次數放大成 500, 1000, 2000 這麼多次以後就會變成不可忽略的差異了。

2. 我們的實作是利用調整優先度來達到排程，但是實際上在我們調整完優先度之後，CPU 也許不會馬上進行 context switch，而是等到真正的 CPU 排程將他們交換，也就是說會讓本來應該被趕出 CPU 的 process 會再多跑一下下，等到真的排程把他趕走。進而造成每個人完成時間會和預期的有一點差異，彼此也不太一樣(0.74 ~ 0.8 的差異)。
3. 我們的電腦背景其實還跑了許多程式，有時候切換到這些程式，我們的 process 就沒有在跑，但是時間還是繼續流逝。也就是說當電腦比較忙的時候，我們測量出來的時間都會比預期再多一點。