

# Report of Operating System Project 1

戴佑全  
黃子賢  
劉彥廷

April 26, 2016

## 1 Design of Child

schedule policy 設為 SCHED\_FIFO 主程式 main 一開始將自己的 priority 設為 98。main 要叫一個 child 跑時，會透過 pipe 告訴 child 要跑幾個 time unit，再將此 child 的 priority 設為 99 child 跑了指定的時間後 會將自己的 priority 設為 97 把主控權還給 main。

## 2 FIFO

### 2.1 Design

資料結構：一開始打算以 link list 來當作 ready queue，但後來想到有給 process 的數量再 加上並沒有用到 link list 可以在中間隨意插入的優點，所以改用 array 當作 ready queue，並有 2 個變數紀錄現在執行的 child 與下一個要被 fork 的 child。

FIFO 是一個很直觀的 scheduler，因為只有一顆 CPU，所以我的作法是 parent 將 child 要跑多少時間跟 child 說，之後將 child 的 priority 提到最高，而 parent 會重新拿到 CPU 有 2 種情況，

1. 因為可能中途要 fork 其他的 child，所以 child 需要在那時將 CPU 還給 parent，這裡使用的方式是將 parent 的 priority 設為 98，換 child 時將 child 提成 99，當 child 要回給 parent 時將自己設為 97，就能完成 context switch 了，而因為此時 child 還沒執行完，只是暫時換 parent，所以在 parent 中現在執行的變數不變，所以 fork 完之後會接續做。
2. 當 child 執行完後，輪到 parent 時，將 parent 中現在執行的變數增加並 wait child 以防有 zombie。所以 parent 的流程是，先檢查是否在 idle (現在執行的 child = 下一個要被 fork 的 child)，如果是則做 for loop 直到下一個 child 被出來，再來是檢查需不需要 fork，因為並非每次輪到 parent 時都要 fork，再來是將 child 需要跑的時間告訴 child，時間為  $\min(\text{下一個要被 fork 的 child 的 ready time}, \text{現在執行的 child 的 execution time})$ 。

### 2.2 Result

#### 1. FIFO.2.txt

P1 3515  
P2 3516  
P3 3517  
P4 3518

[Project1] 3515 1461500526.987120466 1461500691.939705154  
[Project1] 3516 1461500691.939706083 1461500702.258074897  
[Project1] 3517 1461500702.258075945 1461500704.295347989  
[Project1] 3518 1461500704.295349110 1461500706.332647694

## 2.3 Comparison

與理想中的差不多，但是在dmesg稍微有點誤差，原因大概是parent 與child之間context switch造成的，除了priority有保護順序外，因為FIFO並不考慮preemptive，所以只有上一個做完下一個才會做，再加上每次child死亡都會wait所以這也會保護到答案順序的正確。

## 3 SJF

### 3.1 Design

設計跟FIFO差別不大，唯一的差別是在fork完之後要做一次sort，有2種情況，

1. 現在有child正在執行中，那麼只sort除了這支child的其他已被fork出來的child。
2. 現在沒有child正在執行中，那麼sort所有已被fork出來的child。除此之外剩下皆與FIFO相同。

### 3.2 Result

1. SJF\_3.txt

```
P1 2730
P4 2733
P5 2734
P6 2735
P7 2736
P2 2731
P3 2732
P8 2737
```

```
[Project1] 2730 1461506590.620579045 1461506597.056205686
[Project1] 2733 1461506597.056208329 1461506597.077219594
[Project1] 2734 1461506597.077221749 1461506597.098146308
[Project1] 2735 1461506597.098147463 1461506605.898578152
[Project1] 2736 1461506605.898581016 1461506614.651940234
[Project1] 2731 1461506614.651943364 1461506625.677029525
[Project1] 2732 1461506625.677032127 1461506641.046276916
[Project1] 2737 1461506641.046279188 1461506660.897146737
```

## 3.3 Comparison

SJF與FIFO很像，都是不preemptive，所以同FIFO，順序並不太會錯誤，在來就是因為要頻繁的sort所以在時間誤差方面會比較大。

## 4 PSJF

### 4.1 Design

1. Data structure: A poor man's priority queue, implemented by an array and qsort. The smaller one's execution time is, the higher its priority is.
  - (a) Insert:  $O(n \log n)$
  - (b) Pull:  $O(1)$
2. Main while loop:

```

while(there exists non-finished processes) do
  if(clock is the next ready process's ready time) do
    fork this process
    insert this process to priority queue
  else
    wait until next ready process
    continue
  end

  pull the process from the priority queue
  if(next ready process's ready time is smaller than this process's finish time) do
    this process can only run (next ready process's ready time - clock)
  else
    this process can run til it finishes
    wait this child, and mark this process as finished
  end
end
end

```

## 4.2 Result

### 1. PSJF 2.txt

```

P2 3195
P1 3194
P4 3197
P5 3198
P3 3196

```

```

[Project1] 3195 1461556744.671319569 1461556746.818243380
[Project1] 3194 1461556742.541598901 1461556751.107887218
[Project1] 3197 1461556753.257726208 1461556757.596873203
[Project1] 3198 1461556757.596918243 1461556759.755637682
[Project1] 3196 1461556751.107896674 1461556766.169973474

```

## 4.3 Comparison

Basicly, the result is as same as the prediction. Sometimes, the error will be up to 0.2 second. The reason might be that operation system did some content switch to run other applications.

## 5 RR

### 5.1 Design

一開始用array存下所有process資訊並照ready time排序。ready queue用circular linked list實作，指標變數cur\_job指著circular linked list中正要執行的process(也就是ready queue的第一個process。) cur\_job->next是ready queue中下一個要執行的process，cur\_job->last是ready queue中最後一個process。當目前的process的時間用完，要換下一個時，只要cur\_job = cur\_job->next，目前的process就排到ready queue中的最後。當新的process ready，就要排進ready queue的最後，也就是將它塞進cur\_job->last。

while loop每個iteration主要流程: 一開始會檢查是否有process ready，有就fork child。接著如果ready queue是空的(cur\_job == NULL): 就計算要等多久下個process會ready，等完後進入下一個iteration。否則: 決定這次cur\_job要跑多久，跑完後做對應的操作，進入下一個iteration。所有process執行完畢後，跳出while loop，程式結束。

每個iteration中，決定cur\_job要跑多久的判斷和對應的操作: 比較

1. cur\_job這次time quantum剩餘的時間(remain)

2. 下個process ready的時間(`p_arr[next].r_time - cur_time`)
3. `cur_job` 剩餘的執行時間(`cur_job->process->remain.time`)

時間最短的就是這個iteration中`cur_job`要跑的時間 如果選出來的時間是1，跑完後`cur_job`要換下個process(也就是`cur_job->next`) 如果是2，跑完後進入下個iteration就會fork child 如果是3，完後`cur_job`要換下個process且移除目前跑完的process

## 5.2 Result

P3 2653  
P1 2651  
P2 2652  
P6 2656  
P5 2655  
P4 2654

```
[Project1] 2653 1461682248.715317957 1461682276.941976396
[Project1] 2651 1461682242.580521729 1461682278.953463652
[Project1] 2652 1461682245.679939291 1461682280.969540910
[Project1] 2656 1461682256.829051535 1461682297.064796458
[Project1] 2655 1461682254.806290238 1461682301.062410827
[Project1] 2654 1461682252.791221494 1461682303.065176403
```

## 5.3 Comparison

結果大致符合預期，但相距10000 time unit左右可以觀察到0.2秒的誤差，可能是main要處理 children的context switch 時產生的overhead所導致的。

## 6 Contribution of Each Member

戴佑全: system call, RR

黃子賢: child, FIFO, SJF

劉彥廷: main, PSJF