

B06902125 黃柏瑋 OS Project 1 report

1. 設計

1.1 process struct

對於每個process，scheduler會以struct紀錄：

```
1  typedef struct Process{
2      char name[32];
3      int ready_time;
4      int exec_time;
5      pid_t pid;
6  }Process;
```

scheduler沒有用特殊的資料結構存放processes，僅用簡單的陣列而已。

1.2 process clock

process會在第一次被丟上CPU執行時開始計時，並在完成後結束計時並printk。

1.3 scheduler tool

- **P_CPU 與 C_CPU**：
P_CPU 為scheduler在用的CPU，而 C_CPU 是給child process用的。
- **wake_up**：
將process的priority提升至 **SCHED_RR(80)** 或 **SCHED_RR(50)**，並加上flag **SCHED_RESET_ON_FORK**，避免child process被fork出來後，承襲了scheduler的priority並偷跑。
- **block_down**：
將process的priority降至 **SCHED_RR(10)**。
- **guard process**：
當原本在跑的child process結束之後，scheduler可能還來不及算出下一個 **UNIT_TIME** 該由誰跑，**C_CPU** 就先挑了一個不符合scheduler期待的process執行，導致scheduling的誤差。
因此，scheduler派了一個guard process到 **C_CPU** 上，將priority設在wake_up priority與block_down priority之間，原本在跑的child process結束時，**C_CPU** 會先跑guard process而不是那些被block住的child process。
- **last_id**：
紀錄上個執行的child process。若值為-1，代表目前還沒有child process被執行過。
- **curr_id**：
紀錄此UNIT_TIME中需要被執行的child process。若值為-1，代表此輪沒有需要被執行的process。
- **curr_time**：
紀錄目前的時刻，以整數為單位，每過一個UNIT_TIME就會加一。
- **rr_queue**：
適用於policy為RR的時候，紀錄目前process的排隊狀況。

1.4 scheduling

首先，將所有process根據 `ready_time` 由先而後進行排序。

並對scheduler進行以下設置：

1. 將scheduler放在 `P_CPU`，並利用wake_up將priority調至80。
2. fork出guard process，將其放在 `C_CPU`，並利用wake_up將其priority設為50。

當scheduling開始時，scheduler在每個UNIT_TIME中會做以下事情：

1. 檢查有沒有process的 `ready_time` 等於 `curr_time` 到了，若有就fork出一個child process，記錄其pid，並用 `block_down` 將其priority設為10後放至 `C_CPU` 上。如果是policy是RR，要把該process的id加入 `rr_queue` 中。
2. 根據policy挑出 `curr_id`。若有需要context switch，會先利用wake_up將 `curr_id` 的priority調至80，並利用block_down將 `last_id` 調至10。
3. 跑一個UNIT_TIME。
4. 檢查過完一個UNIT_TIME後，剛剛跑的 `curr_id` 是否已經結束(即 `exec_time` 剩下0)。如果是policy是RR，要把該process的id從 `rr_queue` 中移除。

至於scheduler是根據policy如何挑出 `curr_id`？

一開始，我們先預設 `curr_id = last_id`，因為在沒有找出更適合的人選時，`curr_id` 仍會是上一個process承接。

接著，針對不同的policy檢查是否有更適合的人選：

- FIFO
 1. 因為FIFO是non-preemptive，因此只有當 `curr_id` 從缺時(即 `curr_id = -1` 或 `curr_id` 的pid已經不在)才需由第2點判斷新的 `curr_id`。
 2. 用for loop掃過所有的process，挑出pid還在且擁有最小id者(因為已先排序過了)作為新的 `curr_id`。
- RR
 1. 唯有當 `curr_id` 從缺時(即 `curr_id = -1` 或 `curr_id` 的pid已經不在)，或是rr(在RR policy中process剩餘的時間)已經為0，才需由第2點判斷新的 `curr_id`。
 2. 若RR_queue中沒有process在等，`curr_id` 為-1；否則就把 `rr_queue` 中的head移到tail，`curr_id` 改為新的head。
- SJF
 1. 因為SJF是non-preemptive，因此只有當 `curr_id` 從缺時(即 `curr_id = -1` 或 `curr_id` 的pid已經不在)才需由第2點判斷新的 `curr_id`。
 2. 用for loop掃過所有的process，挑出pid還在且擁有最小 `exec_time` 者作為新的 `curr_id`。
- PSJF
 1. 因為PSJF是preemptive，因此無論如何都需由第2點判斷新的 `curr_id`。
 2. 用for loop掃過所有的process，挑出pid還在且擁有最小 `exec_time` 者作為新的 `curr_id`。

2. 核心版本

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

根據<https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872>新增system call及編譯kernel

3. 比較

3.1 數值轉換

從TIME_MEASUREMENT求出的UNIT_TIME，將dmesg file中的數值轉以UNIT_TIME為單位，與理論值進行比較。

$$avg_error = \frac{abs((real_end - real_start) - (theoretical_end - theoretical_start))}{\# \text{ of processes}}$$

3.2 比較

比較幾個test_case中實際跑出來的dmesg file與理論值的差異，其餘的可以至my_output資料夾中查看。為了方便比較，實際值與理論值中最小的開始時間已經對齊。

FIFO_1

```
1 Name real_start real_end theo_start theo_end
2 P1 0.0 476.79 0.0 500.0
3 P2 476.88 998.85 500.0 1000.0
4 P3 998.97 1446.28 1000.0 1500.0
5 P4 1446.35 1934.64 1500.0 2000.0
6 P5 1934.72 2427.95 2000.0 2500.0
7 avg error = 23.270000000000017 unit time
```

FIFO_2

```
1 Name real_start real_end theo_start theo_end
2 P1 0.0 80429.17 0.0 80000.0
3 P2 80429.27 84973.29 80000.0 85000.0
4 P3 85462.18 86474.05 85000.0 86000.0
5 P4 86474.13 87366.86 86000.0 87000.0
6 avg error = 251.07250000000568 unit time
```

RR_3

```
1 Name real_start real_end theo_start theo_end
2 P3 2837.15 16606.75 3000.0 17000.0
3 P1 0.0 18306.09 0.0 18500.0
4 P2 1437.45 18886.68 1500.0 19000.0
5 P6 6696.98 26811.88 7000.0 27000.0
6 P5 5246.67 28641.14 5500.0 29000.0
7 P4 4733.81 29273.58 5000.0 30000.0
8 avg error = 192.62333333333333 unit time
```

RR_4

```

1 | Name real_start real_end theo_start theo_end
2 | P4 1480.05 5399.27 1500.0 5500.0
3 | P5 1973.53 5920.67 2000.0 6000.0
4 | P6 2475.55 6450.07 2500.0 6500.0
5 | P3 968.93 14453.15 1000.0 14500.0
6 | P7 3466.1 18249.03 3500.0 18500.0
7 | P2 469.51 19856.3 500.0 20000.0
8 | P1 0.0 22943.28 0.0 23000.0
9 | avg error = 80.27142857142891 unit time

```

SJF_1

```

1 | Name real_start real_end theo_start theo_end
2 | P2 0.0 1884.14 0.0 2000.0
3 | P3 2050.91 2995.07 2000.0 3000.0
4 | P4 3066.62 7076.55 3000.0 7000.0
5 | P1 7153.82 14138.01 7000.0 14000.0
6 | avg error = 49.35999999999984 unit time

```

SJF_5

```

1 | Name real_start real_end theo_start theo_end
2 | P1 0.0 2020.11 0.0 2000.0
3 | P2 2020.22 2534.56 2000.0 2500.0
4 | P3 2544.1 3050.15 2500.0 3000.0
5 | P4 3071.02 3578.9 3000.0 3500.0
6 | avg error = 12.095000000000027 unit time

```

PSJF_2

```

1 | Name real_start real_end theo_start theo_end
2 | P2 996.15 1984.7 1000.0 2000.0
3 | P1 0.0 4047.59 0.0 4000.0
4 | P4 5135.58 7108.09 5000.0 7000.0
5 | P5 7116.28 8128.51 7000.0 8000.0
6 | P3 4095.02 10840.74 4000.0 11000.0
7 | avg error = 70.60800000000002 unit time

```

PSJF_4

```

1 | Name real_start real_end theo_start theo_end
2 | P3 92.37 1048.61 100.0 1100.0
3 | P2 0.0 2970.17 0.0 3000.0
4 | P4 3001.22 7031.17 3000.0 7000.0
5 | P1 7104.96 14048.8 7000.0 14000.0
6 | avg error = 39.925000000000027 unit time

```

3.3 分析

由上面的結果可知，實際操作priority scheduling還是無法完全符合理論值，以下有幾點可能原因：

1. TIME_UNIT是推估出來的，本身就存在一些差異。

2. 電腦上還會執行許多別的程式，或多或少會瓜分使用CPU的時間；當他們為數不多時結果會更靠近理論值一些，反之則可能背而遠之。
3. 以上的差異在process執行越多unit time時越是顯著，如FIFO_2的第一個process。
4. scheduler在跑time_unit之外，還有其他事情要做，例如檢查需不需要fork process、挑出下一個執行的process、回收結束的process等等，可能使得process在接收scheduler的命令時有時間上的誤差，一旦換process的次數越多，差異就可能越大(如RR_3)。