

RealTime System Project 2 組別: 16

姓名: 黃子軒 高偉傑

學號: R03922133 R03922117

Part1:

在 *sched_simple_rr.c* 這檔案中，我們針對以下 5 個 function 去改:

1. enqueue_task_simple_rr() :

為了將task放到run queue上，我們要在run queue中的list末端新增一個task，並將nr_running加1，也就是目前在run queue上的個數，實作如下:

```
list_add_tail(&(p->simple_rr_list_item), &(rq->simple_rr.queue));  
(rq->simple_rr.nr_running)++;
```

2. dequeue_task_simple_rr()

為了將task從running狀態切換sleeping狀態，我們要在running queue中的list刪除一個task，並將目前在run queue上的task數減1，實作如下:

```
list_del(&(p->simple_rr_list_item));  
(rq->simple_rr.nr_running)--;
```

3. yield_task_simple_rr()

為了將task從running狀態轉換成waiting狀態，我們先從run queue取得目前正在執行的task，接著再將之從run queue中移除，實作如下:

```
struct task_struct *curr = rq->curr;  
requeue_task_simple_rr(rq, curr);
```

4. task_tick_simple_rr()

每經過一個時間，此函數就會被periodic scheduler呼叫，也就是task已經跑了一個simple_rr_time_slice的時間，所以每次需對p->task_time_slice的值

做-1的動作，當p->task_time_slice為零時，表示時間用完了，要繼續配給task
其所需的時間，於是重設p->task_time_slice，並使用set_tsk_need_resched重
設task的flag等來reschedule，最後用requeue_task_simple_rr將此task放至
queue的最尾端，實作如下：

```
if (--p->task_time_slice > 0) return ;  
  
p->task_time_slice = simple_rr_time_slice;  
  
set_tsk_need_resched(p);  
  
requeue_task_simple_rr(rq, p);  
  
return ;
```

5. pick_next_task_simple_rr()

此方法為挑選一個正在waiting狀態的task使之轉換成running狀態，如果
run queue是空的，則回傳NULL，否則用list_first_entry去取得並回傳run queue
中的第一個task，並更新該task的se.exec_start值，也就是schedule entity中紀錄
開始執行的時間。實作如下：

```
if (list_empty(&(rq->simple_rr.queue))) return NULL;  
  
struct task_struct* next_task = list_first_entry(&rq->simple_rr.queue,  
struct task_struct, simple_rr_list_item);  
  
next_task->se.exec_start = rq->clock;  
  
return next_task;
```

實作結果：

3. sched_simple_rr.c

大致上跟 part1 差不多，但多了一句， `task_time_slice` 是 `quantum`，每過一段時間會減 1，當用完時需重新填充為 `weighted_time_slice`。

```
p->task_time_slice = p->weighted_time_slice;
```

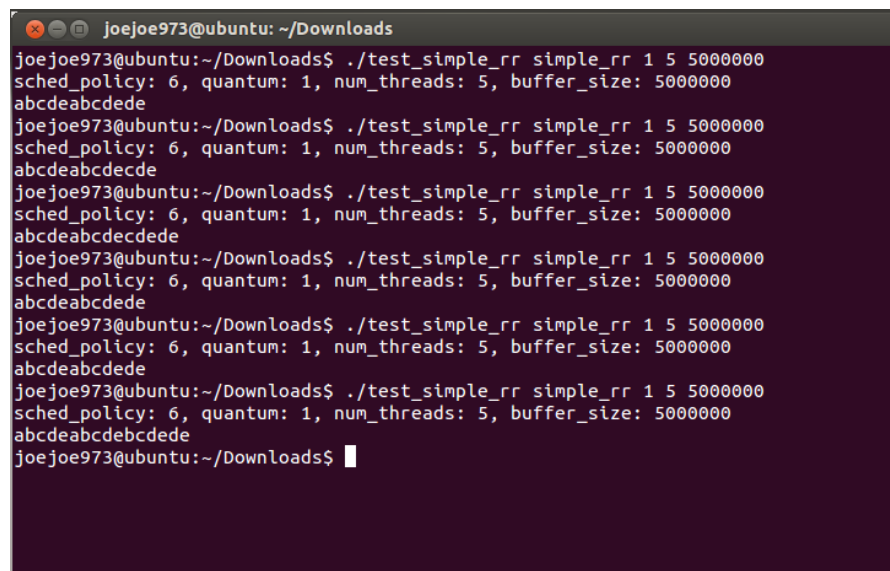
4. test_simple_rr.c

在產生 `num_threads` 的那個 `for` 迴圈裡，必須要多一句且放置於 `pthread_create` 前面:

```
syscall(SYS_simple_rr_setquantum, (quantum * (num_threads - targs->prio + 1)));
```

此用意在於將新給的 `weighted_time_slice` 分配給接下來的 `task`，若優先權越大(`prio` 數字越小)，則分配給的 `time_slice` 將會越大，則越高優先權之工作將會越快完成，反之亦然。

實作結果:



```
joejoe973@ubuntu: ~/Downloads
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcde
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcdecde
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcdecde
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcde
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcde
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcdecde
joejoe973@ubuntu:~/Downloads$
```

不斷地測試 `simple_rr`，`quantum` 令為 1，`buffer` 令為 5000000：結果大致上差不多，優先權較大的 `task` 都會較快完成(`a>b>c>d>e`)，最後幾個完成的都是優先權較低的，符合我們要的結果。

Part3 : Bonus

改良目的:為了創造一個公平且不會starvation的環境，我們希望低優先權的task也能不斷地的被執行到，但是高優先權的task依然能比低優先權task先執行完。

我們改了2份檔案: *sched.h*, *sched_simple_rr.c* :

1. sched.h

為了讓每一輪只會讓每一個task挑到一次，在task_struct裡面新增一個變數: int haspicked，若被挑過則會將此值改成1。

2. sched_simple_rr.c

為了要達到改挑priority最小之目的，我們在pick_next_task_simple_rr裡面，新增了一段code。

首先，先搜尋完整個queue，從還沒被挑過的tasks中挑weighted_time_slice最小者，最後再將它的haspicked值設為1。不過要注意的是，有可能全部都被挑過了，意即haspicked值全部皆為1，則再重新搜尋一次整個queue找出weighted_time_slice最小者，並將所有的haspicked值reset。

3. test_simple_rr.c

為了讓使用者能自行輸入priority順序，但是priority的範圍限制為1到num_threads，我們在這檔案裡面實作了如下:

```
int myprio[10], tmp;

for (i = 0; i < num_threads; i++) {
    tmp = scanf("%d", &myprio[i]);
}
```

實作結果:

