## 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:

```
ineine973@ubuntu: ~/Downloads
     aeaeaaeaeaeae
joejoe973@ubuntu:~/Downloads$ ^C
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 500000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 500000
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 0 5 500000
sched_policy: 6, quantum: 0, num_threads: 5, buffer_size: 500000
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcdeabcdeabcdebdb
joejoe973@ubuntu:~/Downloads$
```

- 1. 測試 default:結果為不斷地 context-switch,需要強制中斷。
- 測試 simple\_rr, quantum 令為 1, buffer 令為 5000000,
   結果為: abcdeabcdeabcdeabcde.....bdb。
- 3. 測試 simple\_rr,quantum 令為 0,buffer 令為 50000 結果為: abcde。

#### Part2:

我們一共改了四份檔案,分別是: sched.c, sched.h, sched\_simple\_rr.c, test\_simple\_rr.c。

#### 1. sched.h

為了實作weighted\_time\_slice這個變數,我們宣告了
"unsigned int weighted\_time\_slice;"。

#### 2. sched.c

在\_sched\_fork函數中,將weighted\_time\_slice初始化為
simple\_rr\_time\_slice,實作如下: p->weighted\_time\_slice =
simple\_rr\_time\_slice;

### 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
abcdeabcdede
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
abcdeabcdecdede
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
labcdeabcdede
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
labcdeabcdede
joejoe973@ubuntu:~/Downloads$ ./test_simple_rr simple_rr 1 5 5000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 5000000
abcdeabcdebcdede
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]);
}</pre>
```

# 實作結果:

```
🔵 📵 jack0602@ubuntu: ~/Desktop
jack0602@ubuntu:~/Desktop$ ./test simple rr simple rr 1 5 90000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 90000000
1 2 3 4 5
12345
jack0602@ubuntu:~/Desktop$ ./test_simple_rr simple_rr 1 5 90000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 90000000
5 4 3 2 1
54321
cabcabcabcababababababababababa
jack0602@ubuntu:~/Desktop$ ./test_simple_rr simple_rr 1 5 90000000
sched_policy: 6, quantum: 1, num_threads: 5, buffer_size: 90000000
2 5 1 3 4
25134
<u>bedbedbebebebebebebebebeb</u>
jack0602@ubuntu:~/Desktop$
```

- 1. 測試 優先權順序為 a>b>c>d>e: 結果為低優先權的 task 會最早做,但是高優先權之 task 依然會先完成。
- 2. 測試 優先權順序為 e>d>c>b>a: 結果會同上。
- 4. 測試 優先權順序為 c>a>d>e>b: 結果會同上,與預期相同。

参考: 助教投影片、Linux Kernel: 強大又好用的 list head 結構

分工: 這次實作先由每個組員各自build出作業所需的kernel,再針對有疑問的地方進行討論並修正,而bonus也是一起提出想法,接著實作,還有跟實驗室的其他同學一起討論,這次實作學了蠻多東西,過程充實且收穫豐富,bonus原本想到的有給task指定的權重或是指定一個超級task必須在任何task之前完成,最後決定實作高偉傑同學的想法-創造一個公平不會starvation的環境使權重低的task也能不斷地執行到。希望助教會喜歡且不吝於給高分,謝謝助教!