

# HW2

## 1. Describe how you implemented the program in detail:

```
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ sudo ./sched_test.sh ./sched_demo ./sched_demo_312512032
Running testcase 0 : ./sched_demo -n 1 -t 0.5 -s NORMAL -p -1
Result: Success!
Running testcase 1 : ./sched_demo -n 2 -t 0.5 -s FIFO,FIFO -p 10,20
Result: Success!
Running testcase 2 : ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Result: Success!
```

(a). 宣告用來同步所有 thread 的 barrier,並定義一個名為 thread\_info 的 struct 用來儲存每個 thread 的資訊:

thread\_id: Thread ID

thread\_number: 第幾個 thread

sched\_policy: 排程策略

sched\_priority: 優先級

time\_wait: busy wait 的時間

```
1 #define _GNU_SOURCE
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <pthread.h>
7 #include <time.h>
8 #include <sched.h>
9 #include <string.h>
10 typedef struct thread_info
11 {
12     pthread_t thread_id;
13     int thread_number;
14     int sched_policy;
15     int sched_priority;
16     double time_wait;
17 } thread_info;
18
19 pthread_barrier_t barrier;
20
```

(b). 使用 getopt 讀取 command-line arguments

```
while ((opt = getopt(argc,argv,"n:t:s:p:"))!=-1){
    switch (opt){
        case 'n':
            num_threads = atoi(optarg);
            //printf("%d\n",num_threads);
            break;
        case 't':
            time_wait = atof(optarg);
            //printf("%f\n",time_wait);
            break;
        case 's':
            policy_temp = malloc(sizeof(char)*(strlen(optarg)+1));
            strcpy(policy_temp,optarg);
            //printf("copy policy!!\n");
            break;
        case 'p':
            priority_temp = malloc(sizeof(char)*(strlen(optarg)+1));
            strcpy(priority_temp,optarg);
            //printf("copy priority!!\n");
            break;
        default:
            break;
    }
}
```

(c). 建立 worker threads 並將對應的 arguments (等待時間、策略跟優先級) 填入。

```
/* 2. Create <num_threads> worker threads */
thread = malloc(sizeof(thread_info)*num_threads);
int i=0;
char_temp=strtok(policy_temp,"");
while(char_temp != NULL){
    thread[i].thread_number=i;
    thread[i].time_wait=time_wait;
    if(strcmp(char_temp,"NORMAL") == 0){
        thread[i].sched_policy=SCHED_OTHER;
        //printf("NORMAL\n");
    }else if ( strcmp(char_temp,"FIFO") == 0){
        thread[i].sched_policy = SCHED_FIFO;
        //printf("FIFO\n");
    }
    char_temp=strtok(NULL,"");
    i++;
}
free(policy_temp);
i=0;
char_temp=strtok(priority_temp,"");
while(char_temp != NULL){
    thread[i].sched_priority = atoi(char_temp);
    //printf("%d\n",atoi(char_temp));
    char_temp = strtok(NULL,"");
    i++;
}
free(priority_temp);
```

(d). 將 process 綁訂到 CPU ID 為 0 的 CPU，以用來讓實現所有 thread 爭奪同一個 CPU 資源。接著初始化 thread 數量加一個 barrier,因為除了要擋住 num\_thread 數量的 thread 以外，還要有一個信號讓他們同步執行所以要在加 1 個。

```
/* 3. Set CPU affinity */
CPU_ZERO(&set);
CPU_SET(0,&set);
sched_setaffinity(getpid(),sizeof(set),&set);

pthread_barrier_init(&barrier,NULL,num_threads+1);
```

(e).根據條件設定每個 thread 的屬性,若是 SCHED\_FIFO 設定其優先級及策略,若是 SCHED\_OTHER 則設定其策略:

pthread\_attr\_init: 初始化 thread 的屬性。

pthread\_attr\_setinheritsched: 設定 thread 是否繼承 main process 的屬性設定。

pthread\_attr\_setschedpolicy: 設定排程策略。

param.sched\_priority: 若是 SCHED\_FIFO,則設定其優先級。

pthread\_attr\_setschedparam: 設定 thread 的屬性(優先級及策略)。

pthread\_create: 使用設定好的屬性創建 thread。

pthread\_attr\_destroy: 設定好後釋放屬性資源。

```
for (int i = 0; i < num_threads; i++) {
    //printf("thread %d :",thread[i].thread_number);
    //printf("policy %d ,",thread[i].sched_policy);
    //printf("priority %d \n",thread[i].sched_priority);
    /*4. Set the attributes to each thread */

    pthread_attr_init(&attr);
    pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);

    if (thread[i].sched_policy == SCHED_FIFO){

        pthread_attr_setschedpolicy(&attr,SCHED_FIFO);

        param.sched_priority=thread[i].sched_priority;
        pthread_attr_setschedparam(&attr,&param);

        pthread_create(&thread[i].thread_id,&attr,thread_func,(void*)(thread+i));
    }
    else if (thread[i].sched_policy == SCHED_OTHER){

        pthread_attr_setschedpolicy(&attr,SCHED_OTHER);
        pthread_create(&thread[i].thread_id,&attr,thread_func,(void*)(thread+i));
    }
    pthread_attr_destroy(&attr);
}
```

(f). 在所有的 thread 設定好後使用 pthread\_barrier\_wait,使得 barrier 到達設定數量，讓所有 thread 同時啟動，根據他們的排程策略及優先級搶奪 CPU 資源。最後當所有的 thread 都完成動作時釋放資源。

```
/* 5. Start all threads at once */
pthread_barrier_wait(&barrier);
/* 6. Wait for all threads to finish */
for (int i = 0; i < num_threads; i++) {
    pthread_join(thread[i].thread_id,NULL);
}
pthread_barrier_destroy(&barrier);
free(thread);
return 0;
```

(g).每個 thread 開始時先被 barrier 阻擋，直到所有 thread 準備好，接著根據資訊顯示自己是哪個 thread，並 busy-wait 指定秒數，重複三次。busy-waiting 的實現方式是記錄此 thread 占用 CPU 的起始時間，並不斷確認此 thread 占用 CPU 的當前時間是否經過指定的時間長度。

```
void* thread_func(void* arg){
    pthread_barrier_wait(&barrier);
    thread_info * thread = (thread_info *)arg;
    //printf("which cpu= %d \n",sched_getcpu());
    for (int i=0 ; i<3 ; i++){
        struct timespec start,end ;
        clock_gettime(CLOCK_THREAD_CPUTIME_ID,&start);
        clock_gettime(CLOCK_THREAD_CPUTIME_ID,&end);
        printf("Thread %d is starting\n", thread->thread_number);

        while ((end.tv_sec- start.tv_sec) + (end.tv_nsec-start.tv_nsec)*1e-9 <= thread->time_wait)
        {
            //busy-wait
            clock_gettime(CLOCK_THREAD_CPUTIME_ID,&end);
        }
    }
    pthread_exit(NULL);
}
```

2. Describe the results of `sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30`, and what causes that.

**Case 1 ( `sched_rt_runtime_us == sched_rt_period_us` ) :**

由參考資料[1]，可知 `SCHED_OTHER(SCHED_NORMAL)` 其 priority 默認為 0 因此可以把 Thread 0 的優先級當作 0，而 `SCHED_FIFO` 根據優先級(static priorities higher then 0)決定執行順序,高優先級者有優先執行，因此 Thread 2 (priority 30) 先執行，接著是 Thread 2 (priority 10),最後是 Thread 0 (priority 0),且因為 `sched_rt_runtime_us` 與 `sched_rt_period_us` 相同，因此在 `SCHED_FIFO` 結束前 `SCHED_OTHER` 無法占用 CUP。(如圖 2-1)

```
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_runtime_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_period_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Thread 2 is starting
Thread 2 is starting
Thread 2 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 0 is starting
Thread 0 is starting
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$
```

圖 2-1

**Case 2 ( `sched_rt_runtime_us < sched_rt_period_us` ) :**

由於現在 `sched_rt_runtime_us` = 950000 (0.95 秒),而 `sched_rt_period_us` = 1000000 (1 秒),也就是說在一個周期內(1 秒),有 0.95 秒的時間是給 real-time task (`SCHED_FIFO`),而剩下 0.05 秒是給普通的 task (`SCHED_OTHER`),因此執行時不會像 CASE 1 一樣完全由 real-time task 占用 CPU。

先由 Thread 2 (priority 30) 先取得 並運行 0.95 秒後，再由 CFS 分配剩下的 0.05 秒給 `SCHED_OTHER` 的 task(只有一個 task 所以這 0.05 秒都給他),再根據優先級執行 0.95 秒，然後再讓 `SCHED_OTHER` 的 task 執行 0.05 秒，周而復始，直到 task 完成。(如圖 2-2)

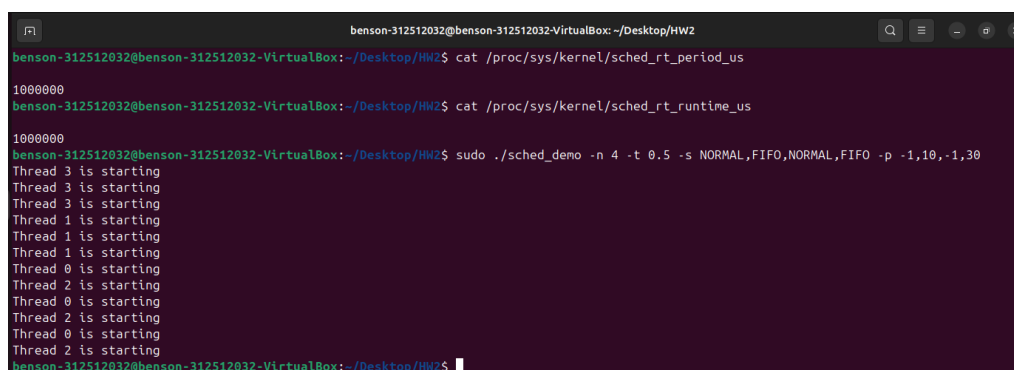
```
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_runtime_us
950000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_period_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 2 is starting
Thread 2 is starting
Thread 1 is starting
Thread 0 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$
```

圖 2-2

3. Describe the results of `sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30`, and what causes that.

**Case 1 ( `sched_rt_runtime_us == sched_rt_period_us` ) :**

因為 `sched_rt_runtime_us` 與 `sched_rt_period_us` 相同, 因此在 `SCHED_FIFO` 結束前 `SCHED_OTHER` 無法占用 CUP, 因此先由優先級高的 Thread 3 先取得資源, 接著由優先級次高的 Thread 1 取得資源, 待所有 real-time task 完成後, 透過 CFS 將資源公平的分配給 `SCHED_OTHER` 的 Thread 0 與 Thread 2。(圖 3-1)



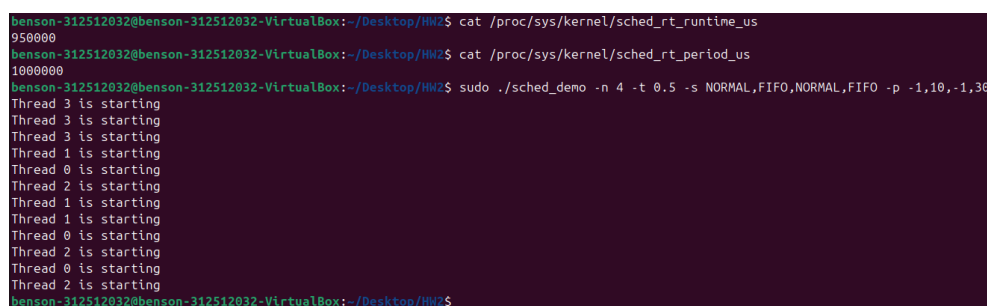
```
benson-312512032@benson-312512032-VirtualBox: ~/Desktop/HW2
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_period_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_runtime_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is starting
Thread 3 is starting
Thread 3 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$
```

圖 3-1

**Case 2 ( `sched_rt_runtime_us < sched_rt_period_us` ) :**

由於現在 `sched_rt_runtime_us` = 950000 (0.95 秒), 而 `sched_rt_period_us` = 1000000 (1 秒), 也就是說在一個周期內(1 秒), 有 0.95 秒的時間是給 real-time task (`SCHED_FIFO`), 而剩下 0.05 秒是給普通的 task (`SCHED_OTHER`), 因此執行時不會像 CASE 1 一樣完全由 real-time task 占用 CPU。

因此先由 Thread 3 (priority 30) 先取得 並運行 0.95 秒後, 再由 CFS 分配剩下的 0.05 秒給 `SCHED_OTHER` 的 task (Thread 0 與 Thread 2 公平分配此 0.05 秒), 再根據優先級給 `SCHED_FIFO` 執行 0.95 秒, 然後再讓 `SCHED_OTHER` 的 task 執行 0.05 秒, 周而復始, 直到所有 task 完成。(如圖 3-2)



```
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_runtime_us
950000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ cat /proc/sys/kernel/sched_rt_period_us
1000000
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$ sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is starting
Thread 3 is starting
Thread 3 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
benson-312512032@benson-312512032-VirtualBox:~/Desktop/HW2$
```

圖 3-2

#### 4. Describe how did you implement n-second-busy-waiting?

使用 `clock_gettime()` 紀錄起始的時間(start),而 `CLOCK_THREAD_CPUTIME_ID` 指的是 Thread 在 CPU 上執行的時間,並使用 while 迴圈 讓 Thread 實現 busy-waiting,當等待時間未達到指定長度(n-second)時繼續 while 內讀取當前 Thread 在 CPU 上執行的時間(end),直到等待時間符合指定長度,如圖四。

`Sleep()`無法滿足的原因在於,他會把當前的 thread 暫時釋出 CPU 資源,這樣的話資源就會被其他高優先級的 Thread 拿走,直到 `sleep()`結束,因此不滿足 busy-waiting 的要求。

```
struct timespec start,end ;
clock_gettime(CLOCK_THREAD_CPUTIME_ID,&start);
clock_gettime(CLOCK_THREAD_CPUTIME_ID,&end);
printf("Thread %d is starting\n", thread->thread_number);

while ((end.tv_sec- start.tv_sec) + (end.tv_nsec-start.tv_nsec)*1e-9 <= thread->time_wait)
{
    //busy-wait
    clock_gettime(CLOCK_THREAD_CPUTIME_ID,&end);
}
```

圖四

#### 5. What does the *kernel.sched\_rt\_runtime\_us* effect? If this setting is changed, what will happen?

由題目 3 及題目 4 的 case 1 、case 2 可以得知我們可以透過調整 `kernel.sched_rt_runtime_us`,來控制一個周期內(`kernel.sched_rt_period_us`), Real-time task (`SCHED_FIFO`)和 Normal task (`SCHED_OTHER`)所占用的比例,像是 圖 2-1、圖 3-1 中,由於 `kernel.sched_rt_runtime_us` 等於 `kernel.sched_rt_period_us`,因此需等所有的 Real-time task (`SCHED_FIFO`)完成後 Normal task (`SCHED_OTHER`) 才能去根據 CFS 分配資源。

而當 `kernel.sched_rt_runtime_us` 不等於 `kernel.sched_rt_period_us` 時,如圖 2-2、2-3,CPU 資源會將一個週期 `kernel.sched_rt_period_us` 中 `kernel.sched_rt_runtime_us` 的時間分配給 Real-time task (`SCHED_FIFO`),即使 Real-time task (`SCHED_FIFO`)還未完成,但時間到了(0.95 秒),系統還是會把剩下的時間(0.05 秒)先給 Normal task (`SCHED_OTHER`),然後再切回 Real-time task (`SCHED_FIFO`),周而復始。

這樣的設計是為了防止 Real-time task (`SCHED_FIFO`)壟斷 CPU 導致 Normal task (`SCHED_OTHER`)沒有機會去執行。且我們可以根據應用需求去調整 `kernel.sched_rt_runtime_us` 讓系統可以在 REAL-TIME 跟公平 之間去協調。

#### 參考資料:

- [1]. sched(7) — Linux manual page , <https://man7.org/linux/man-pages/man7/sched.7.html>
- [2]. Clock\_gettime , [https://linux.die.net/man/3/clock\\_gettime](https://linux.die.net/man/3/clock_gettime)

[3]. Scheduling - RT throttling, [https://wiki.linuxfoundation.org/realtime/documentation/technical\\_basics/sched\\_rt\\_throttling](https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_rt_throttling)