

1. Describe how you implemented the program in detail. (10%)

```
1  #define _GNU_SOURCE
2  #include <sched.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <string.h>
7  #include <time.h>
8  #include <pthread.h>
9
10 pthread_barrier_t barrier;
11 typedef struct {
12     int id;
13     char policy[10];
14     int priority;
15     float exec_time;
16 } thread_info_t;
```

#define _GNU_SOURCE

啟用 GNU 擴展功能，以便使用非標準的函式，例 sched_setaffinity。

<sched.h>：提供與排程相關的函式和定義（如 sched_setaffinity & SCHED_FIFO 等）。

<stdio.h>：用於標準輸出與輸入（如 printf）。

<stdlib.h>：提供記憶體分配和標準函式（如 malloc, atoi 等）。

<unistd.h>：支援 POSIX 系統函式（如 getopt）。

<string.h>：處理字串操作（如 strtok, strcpy 等）。

Struct thread_info_t

這個結構體用於儲存每個執行緒需要的基本資訊：

id：第幾個 thread

policy：執行 thread 的排程策略（例如 FIFO, RR, NORMAL）。

priority：執行緒的優先級（適用於 FIFO 和 RR strategy）。

exec_time：模擬 busy waiting 的執行時間，單位為秒。

```
void *worker_thread(void *arg) {  
    thread_info_t *info = (thread_info_t *)arg;  
    cpu_set_t cpuset;  
    CPU_ZERO(&cpuset);           // clear all CPUs  
    CPU_SET(0, &cpuset);  
    if (sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) == -1) {  
        perror("sched_setaffinity");  
        return NULL;  
    }           // Set CPU 0 (assuming all threads run on CPU 0)  
    pthread_barrier_wait(&barrier);  
    // Simulate busy waiting  
    for (int i = 0; i < 3; i++) {  
        printf("Thread %d is starting\n", info->id);  
        volatile double result = 0.0; // Use volatile to prevent compiler optimization  
        double end_time = info->exec_time;  
        double elapsed_time = 0.0;  
        struct timespec start, current;  
  
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start);  
  
        while (elapsed_time < end_time) {  
            // Perform simple calculations to simulate busy waiting  
            for (int i = 0; i < 1000; i++) {  
                result += i * 0.0001;  
            }  
  
            clock_gettime(CLOCK_THREAD_CPUTIME_ID, &current);  
            elapsed_time = (current.tv_sec - start.tv_sec) +  
                (current.tv_nsec - start.tv_nsec) / 1e9;  
        }  
    }  
    return NULL;  
}
```

每個 thread 執行的 function，先使用 sched_setaffinity 將所有的 threads 固定在同一個 CPU core 上運行，接著使用 pthread_barrier_wait 確保所有 threads 是同時開始執行的，同時 barrier 須為全域變數確保所有 thread 都能看到他。

接著每個 thread 會模擬 busy waiting 3 次，並透過在 for loop 中進行簡單計算

來占用 CPU，直到執行時間到指定的 `exec_time` 為止，`clock_gettime` 的 arguments 有五個選項，分別是 `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID`。

分別對應到的是系統時鐘的實時時間(wall clock time)，系統從啟動以來到目前為止的時間，一個 Process 占用 CPU 的總時間(所有 thread 加總，計時是 for 一個 process)，一個 Thread 占用 CPU 的時間。根據本次 assignment 的設定，需要計時每個 thread 占用 CPU 的時間，並且在一個 thread 被其他 thread preempt 的時候會停止計時，等到這個 thread 回到 CPU 執行時才會恢復計時，所以我選擇了 `CLOCK_THREAD_CPUTIME_ID` 完成 assignment。

```
int main(int argc, char *argv[]) {
    int num_threads = 0;
    float exec_time = 0;
    char *schedules = NULL;
    char *priorities = NULL;
    int opt;
    // Parse command-line arguments
    while ((opt = getopt(argc, argv, "n:t:s:p:")) != -1) {
        switch (opt) {
            case 'n':
                num_threads = atoi(optarg);
                break;
            case 't':
                exec_time = atof(optarg);
                break;
            case 's':
                schedules = strdup(optarg);
                break;
            case 'p':
                priorities = strdup(optarg);
                break;
            default:
                fprintf(stderr, "Usage: %s -n <num_threads> -t <time> -s <policies> -p <priorities>\n", argv[0]);
                exit(EXIT_FAILURE);
        }
    }

    if (num_threads <= 0 || exec_time <= 0 || !schedules || !priorities) {
        fprintf(stderr, "Invalid arguments\n");
        exit(EXIT_FAILURE);
    }
}
```

Main function 的部分先進行 parse，使用 `getopt` 和 `switch` 來獲得每一個

thread 的 argument，四個 arguments 分別是第幾個(n)，執行時間(time, t)，
排程策略(scheduling strategy, s)，優先級(priority, p)。分別使用 atoi, atof,
strup 來傳遞到 main function 的 num_threads, exec_time, schedules,
priorities 四個變數內。(剩下 default 和 if 判斷式為 error handler 防止錯誤的傳
遞變數到這支程式內)

```
pthread_t threads[num_threads];
pthread_attr_t attr;
thread_info_t thread_info[num_threads];
char *sched_tokens[num_threads];
char *prio_tokens[num_threads];
int i;

// Split scheduling policies and priorities
char *token = strtok(schedules, ",");
for (i = 0; i < num_threads && token != NULL; i++) {
    sched_tokens[i] = token;
    token = strtok(NULL, ",");
}
if (i != num_threads) {
    fprintf(stderr, "Mismatch in number of scheduling policies\n");
    exit(EXIT_FAILURE);
}

token = strtok(priorities, ",");
for (i = 0; i < num_threads && token != NULL; i++) {
    prio_tokens[i] = token;
    token = strtok(NULL, ",");
}
if (i != num_threads) {
    fprintf(stderr, "Mismatch in number of priorities\n");
    exit(EXIT_FAILURE);
}
```

接著是把各個 thread 所需要的資料分割出來，使用 strtok 來切分前面以逗點(,) 進行分隔的字串，在第一次切分以後，每呼叫一次繼續往下切，並存入 sched_tokens, prio_tokens 兩個 array 內。

```

pthread_attr_init(&attr);
pthread_barrier_init(&barrier, NULL, num_threads);

for (i = 0; i < num_threads; i++) {
    thread_info[i].id = i;
    strcpy(thread_info[i].policy, sched_tokens[i]);
    thread_info[i].priority = atoi(prio_tokens[i]);
    thread_info[i].exec_time = exec_time;

    struct sched_param param;

    if (strcmp(thread_info[i].policy, "FIFO") == 0) {
        pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    } else if (strcmp(thread_info[i].policy, "RR") == 0) {
        pthread_attr_setschedpolicy(&attr, SCHED_RR);
    } else {
        pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    }

    if (thread_info[i].priority != -1) {
        param.sched_priority = thread_info[i].priority;
        pthread_attr_setschedparam(&attr, &param);
        pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    } else {
        pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED);
    }

    // Create threads
    if (pthread_create(&threads[i], &attr, worker_thread, &thread_info[i]) != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }
}

```

先初始化 threads 和 barrier，然後開始一個循環，對每個 thread 進行初始化和創建。

a. thread_info[i].id = i; :

為每個 thread 設置唯一的 id，這樣可以在執行過程中區分不同的 thread。

b. strcpy(thread_info[i].policy, sched_tokens[i]); :

將對應的 scheduling policy（例如"FIFO"、"RR"）存到 thread_info[i].policy 內。

c. thread_info[i].priority = atoi(prio_tokens[i]); :

將 Priority 轉換為整數並存儲在 thread_info[i].priority 中。

d. thread_info[i].exec_time = exec_time; :

設置 thread 的執行時間（exec_time），將在後面的 busy waiting 過程中使用。

e. struct sched_param param; :

創建一個 sched_param structure，用來設置 thread 的調度優先級。這個 structure 的內容會在後面被傳遞給 pthread_attr_setschedparam 來設定優先級。

第一組 If 判斷式內的程式碼用來設定 thread 的 scheduling policy，根據不同的 policy(FIFO, RR, OTHER)會將 pthread_attr_setschedpolicy 設定為指定的值，第二組則設定優先級別(如果沒有給定則跟隨上一個 thread)。然後用 pthread_create 生成 threads，使用 attr 中設定的屬性(e.g. scheduling policy, priority)，並且將 threads_info 傳遞給該函式。

```
// wait for all threads to finish
for (i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
}

// Cleanup
pthread_attr_destroy(&attr);
pthread_barrier_destroy(&barrier);
free(schedules);
free(priorities);

//printf("All threads finished\n");
return 0;
}
```

接下來 main thread 等待所有創建出的 thread 執行結束。pthread_join 會使 CPU 保持阻塞(Blocking)直到所有的 thread 完成，main thread 才會繼續運行。

最後一步是清理&釋放資源。(destroy, free)

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.

(1)使用 default(`kernel.sched_rt_runtime_us=950000`)的結果：

```
shwan@Shwan:~/Desktop$ sudo ./sched_demo_312512011 -n 3 -t 1.0 -s NORMAL,FIFO,FI
FO -p -1,10,30
[sudo] shwan 的密碼：
Thread 2 is starting
Thread 2 is starting
Thread 0 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
```

可以看到 thread 2 具有最高優先級別&FIFO 故先執行，但執行到中間觸發 realtime throttling mechanism(realtime task 合計超過 0.95s)故所有 realtime thread 都不能跑(1,2)→先跑 thread 0.，接著回去跑 thread 2，處理完任務輪到 thread 1，接著 realtime task 結束換成 normal thread 執行。

(2)將 `kernel.sched_rt_runtime_us=950000 = -1` 的結果：

```
shwan@Shwan:~/Desktop$ time sudo ./sched_demo_312512011 -n 3 -t 1.0 -s NORMAL,
FO,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

real    0m9.024s
user    0m0.009s
sys     0m0.010s
```

thread 2 具有 FIFO+最高優先級(30)故先執行，此時 RT throttling 機制被關閉，所以 thread 2(P=30)和 1(P=10)可以不斷佔有 CPU 直到 realtime task 全數執行完畢才輪到 normal 的 thread 0。Runtime 計時： $\text{num_threads} \times t \times 3 = 9.00\text{s}$ (實際 9.024s)

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. (10%)

(1)使用 default(`kernel.sched_rt_runtime_us=950000`)的結果：

```
shwan@Shwan:~/Desktop$ time 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 0 is starting
Thread 2 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
```

上圖可以看到 thread3 具有最高優先級別&FIFO 故先執行，但執行到中間觸發 realtime throttling mechanism(realtime task 合計超過 0.95s)故所有 realtime thread 都不能跑(thread 1,3)→先跑 thread 0(此時先跑 2 還是先跑 0 取決於機率，不太一定)，接著去跑 thread2(同樣應為觸發 RT throttling)，RT throttling 解除後續跑 realtime 優先級較高的 thread 3，接著是 thread 1，最後是 normal threads(0 和 2)

(2)將 `kernel.sched_rt_runtime_us=950000 = -1` 的結果：

```
shwan@Shwan:~/Desktop$ time sudo ./sched_demo_312512011 -n 4 -t 0.5 -s NORMAL,FI
FO,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 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting

real    0m6.029s
user    0m0.007s
sys     0m0.011s
```


和上一題同樣可以觀察到，在 RT throttling 被關閉的情況下，realtime task 可以無限制占用 CPU，取決於各自優先級別(此處 3>1)，RT task 跑完才會處理 normal process(0&2)，接著由於兩者優先級別相同所以交互執行直到全部結束。Runtime 計時： $\text{num_threads} * t * 3 = 6.00\text{s}$ (實際 6.029s)

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

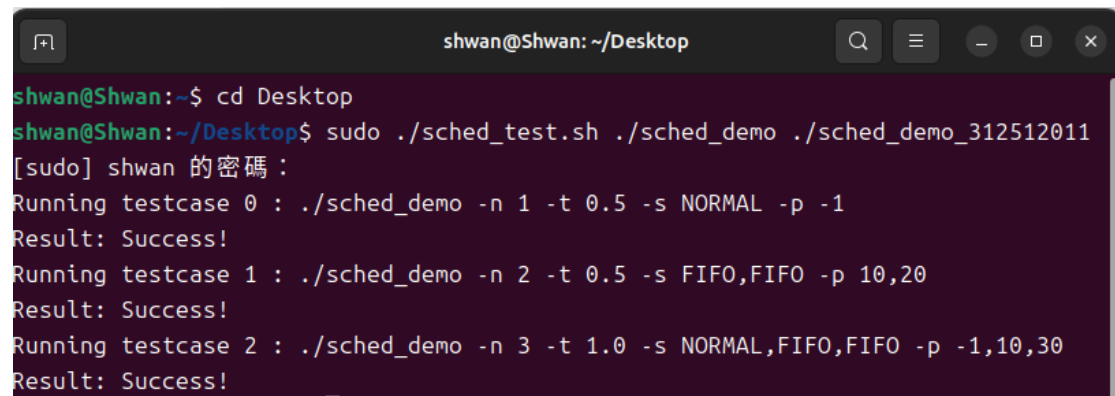
- 開始計時: `clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start)`
- 取得目前執行時間: `clock_gettime(CLOCK_THREAD_CPUTIME_ID, ¤t)`
- 計算經過時間， $\text{elapsed_time} = (\text{current.tv_sec} - \text{start.tv_sec}) + (\text{current.tv_nsec} - \text{start.tv_nsec}) / 1\text{e}9$; (由於 nsec 和 sec 單位不同故要進行處理)
- `while (elapsed_time < end_time)` 判斷是否超過執行時間，如果沒有超過則繼續，超過則跳出此迴圈

5. What does the kernel.sched_rt_runtime_us effect? If this setting is changed, what will happen?

kernel.sched_rt_runtime_us 是一個 Kernel variable，其 default value 是 950000 us，預設每 1 秒為周期，會計算接下來 CPU 是否會有 realtime task 總計超過 0.95 秒。而 realtime threads 和 normal threads 的類別不同，基於 Real-time throttling 的機制，當 realtime threads 的 runtime 總計跑超過這個時間 (0.95 秒)，就會觸發上述機制強制將 realtime thread 停掉，跑 normal process。若將此值設定為 -1 則意味著取消 throttling 機制，task 可以無限制占用 CPU 直到他完成，將此值設定到 -1 在真實電腦中會是不安全的操作。

```
shwan@Shwan:~/Desktop$ cat /proc/sys/kernel/sched_rt_runtime_us
950000
```

附錄：Given test case result

A terminal window titled 'shwan@Shwan: ~/Desktop' with standard window controls. The terminal shows a series of commands and their outputs. The user navigates to the Desktop directory and runs a script with three test cases. Each test case is executed successfully, as indicated by the 'Result: Success!' message. The test cases involve running './sched_demo' with different parameters for number of processes (-n), time (-t), scheduling policy (-s), and priority (-p).

```
shwan@Shwan:~$ cd Desktop
shwan@Shwan:~/Desktop$ sudo ./sched_test.sh ./sched_demo ./sched_demo_312512011
[sudo] shwan 的密碼：
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!
```

從結果顯示程式執行成功，每個測資都有成功執行。