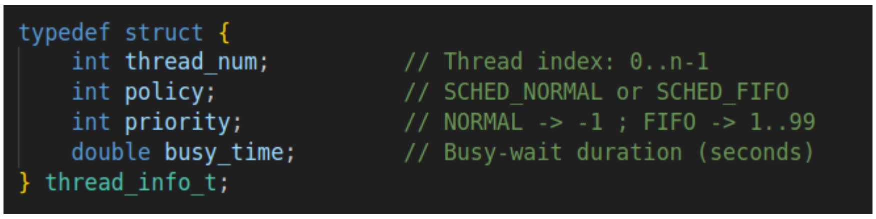
NYCU Operating System - Assignment II

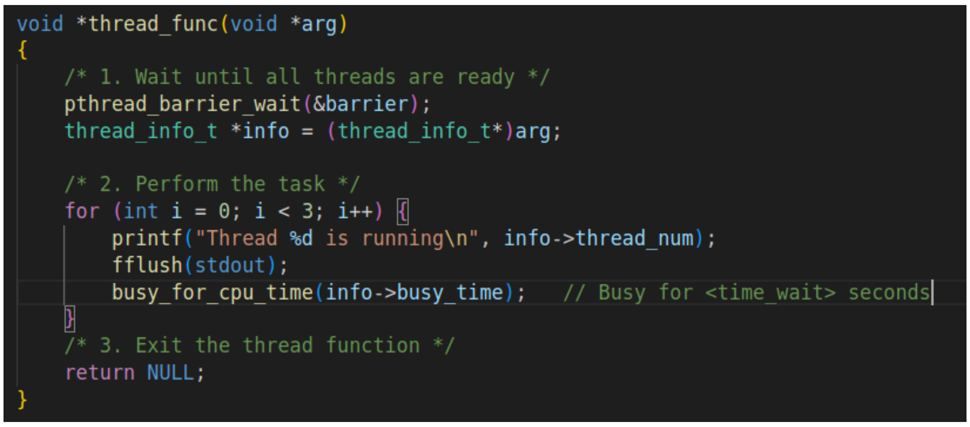
111550129, 林彥亨

1. Describe how you implemented the program in detail. (10%)
    a. 宣告一個 structure 去存 thread information



```
typedef struct {
    int thread_num;          // Thread index: 0..n-1
    int policy;              // SCHED_NORMAL or SCHED_FIFO
    int priority;            // NORMAL -> -1 ; FIFO -> 1..99
    double busy_time;        // Busy-wait duration (seconds)
} thread_info_t;
```

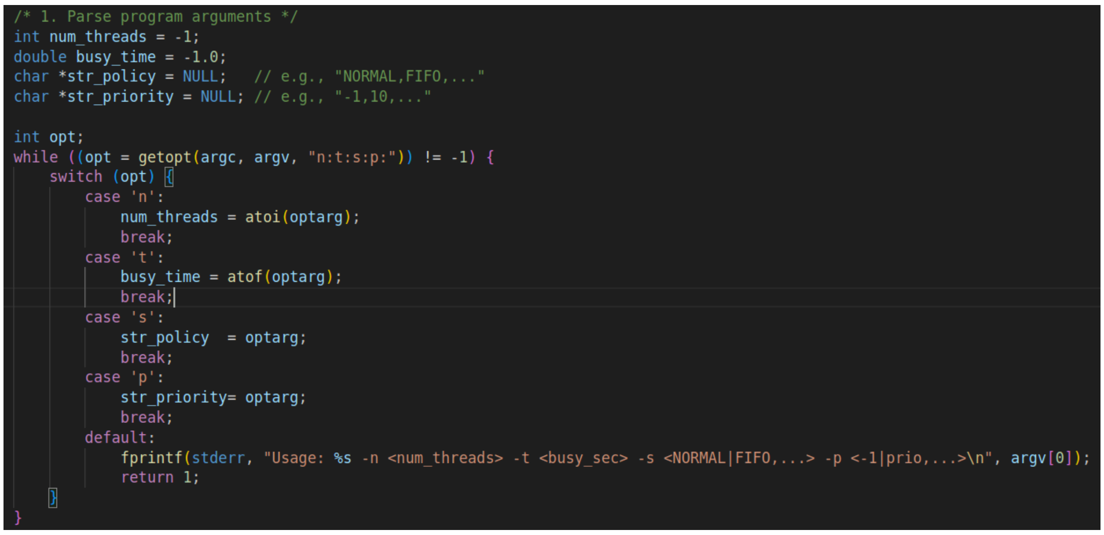
    b. thread_func()
        - 同步每條 worker threads `pthread_barrier_wait(&barrier);`
        - 讓所有 worker thread 在同一個 barrier 等待，直到 main thread 也到達後同時釋放，確保開始時刻一致，避免建立順序導致的偏差。
        - 執行迴圈 `printf("Thread %d is running\n", info->thread_num);`
        - 使用 `busy_for_cpu_time(info->busy_time);` 去量測該thread 的 CPU 使用時間。
        - 回傳 NULL 結束執行緒。



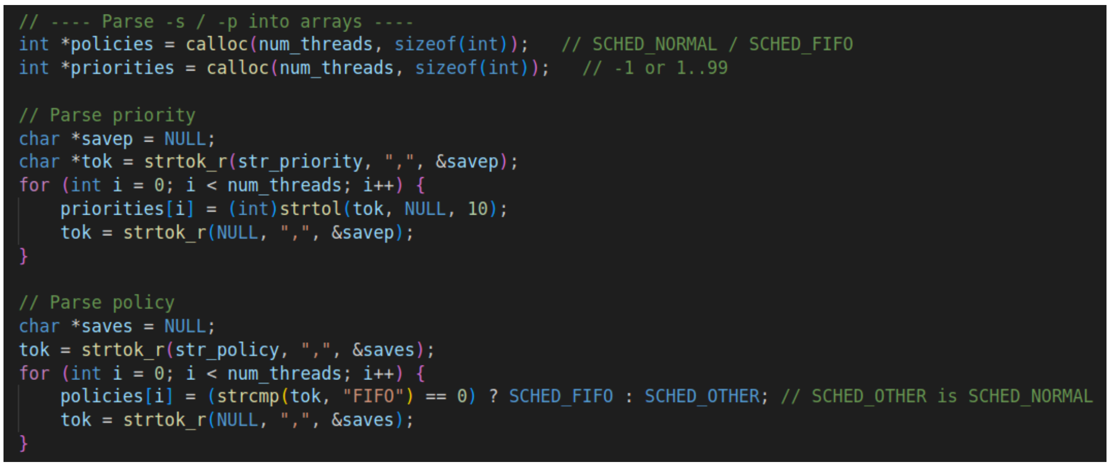
```
void *thread_func(void *arg)
{
    /* 1. Wait until all threads are ready */
    pthread_barrier_wait(&barrier);
    thread_info_t *info = (thread_info_t*)arg;

    /* 2. Perform the task */
    for (int i = 0; i < 3; i++) {
        printf("Thread %d is running\n", info->thread_num);
        fflush(stdout);
        busy_for_cpu_time(info->busy_time);   // Busy for <time_wait> seconds
    }
    /* 3. Exit the thread function */
    return NULL;
}
```

    c. main() - Parse program arguments
        - 使用 `getopt(argc, argv, "n:t:s:p:")` 解析：
            - n：執行緒數量（num_threads）
            - t：每回合忙等秒數（busy_time）
            - s：排程政策清單（字串，如 NORMAL,FIFO,...）
            - p：對應優先權清單（如 -1,10,...）
        - 將 -s/-p 的逗號分隔字串用 strtok_r 解析成陣列：
            - `policies[i] = (strcmp(tok, "FIFO") == 0) ? SCHED_FIFO : SCHED_OTHER;`

○ `priorities[i] = (int)strtol(tok, NULL, 10);`



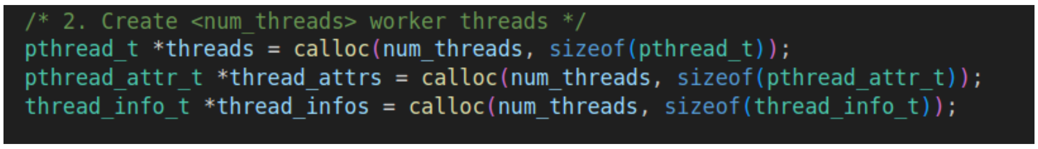
```c
/* 1. Parse program arguments */
int num_threads = -1;
double busy_time = -1.0;
char *str_policy = NULL;   // e.g., "NORMAL,FIFO,..."
char *str_priority = NULL; // e.g., "-1,10,..."

int opt;
while ((opt = getopt(argc, argv, "n:t:s:p:")) != -1) {
    switch (opt) {
        case 'n':
            num_threads = atoi(optarg);
            break;
        case 't':
            busy_time = atof(optarg);
            break;
        case 's':
            str_policy  = optarg;
            break;
        case 'p':
            str_priority= optarg;
            break;
        default:
            fprintf(stderr, "Usage: %s -n <num_threads> -t <busy_sec> -s <NORMAL|FIFO,...> -p <-1|prio,...>\n", argv[0]);
            return 1;
    }
}
```



```c
// ---- Parse -s / -p into arrays ----
int *policies = calloc(num_threads, sizeof(int));   // SCHED_NORMAL / SCHED_FIFO
int *priorities = calloc(num_threads, sizeof(int));   // -1 or 1..99

// Parse priority
char *savep = NULL;
char *tok = strtok_r(str_priority, ",", &savep);
for (int i = 0; i < num_threads; i++) {
    priorities[i] = (int)strtol(tok, NULL, 10);
    tok = strtok_r(NULL, ",", &savep);
}

// Parse policy
char *saves = NULL;
tok = strtok_r(str_policy, ",", &saves);
for (int i = 0; i < num_threads; i++) {
    policies[i] = (strcmp(tok, "FIFO") == 0) ? SCHED_FIFO : SCHED_OTHER; // SCHED_OTHER is SCHED_NORMAL
    tok = strtok_r(NULL, ",", &saves);
}
```

d. main() - Create <num_threads> worker threads

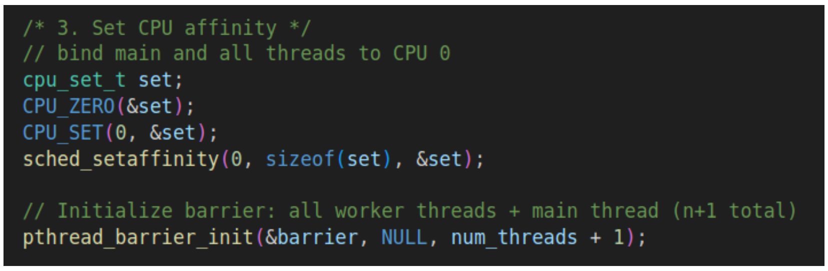
- 準備好存放執行緒識別、屬性、以及傳入函式的參數。



```c
/* 2. Create <num_threads> worker threads */
pthread_t *threads = calloc(num_threads, sizeof(pthread_t));
pthread_attr_t *thread_attrs = calloc(num_threads, sizeof(pthread_attr_t));
thread_info_t *thread_infos = calloc(num_threads, sizeof(thread_info_t));
```

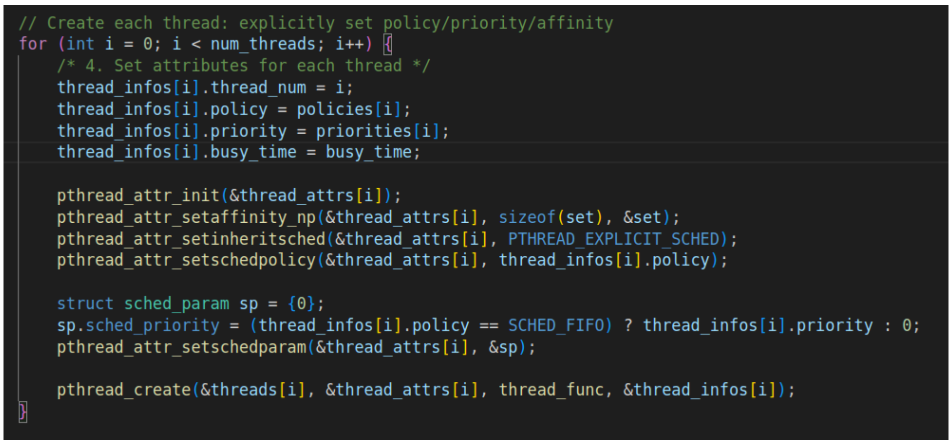
e. main() - Set CPU affinity

- 把主執行緒與所有工人執行緒鎖在同一顆 CPU（CPU 0）。



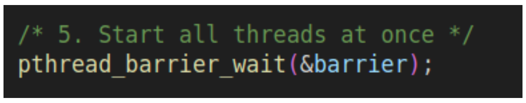
```c
/* 3. Set CPU affinity */
// bind main and all threads to CPU 0
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(0, &set);
sched_setaffinity(0, sizeof(set), &set);

// Initialize barrier: all worker threads + main thread (n+1 total)
pthread_barrier_init(&barrier, NULL, num_threads + 1);
```

f. main() - Set attributes for each thread
- 明確告訴系統每條執行緒要用什麼排程政策與優先權，且不要繼承主執行緒設定。



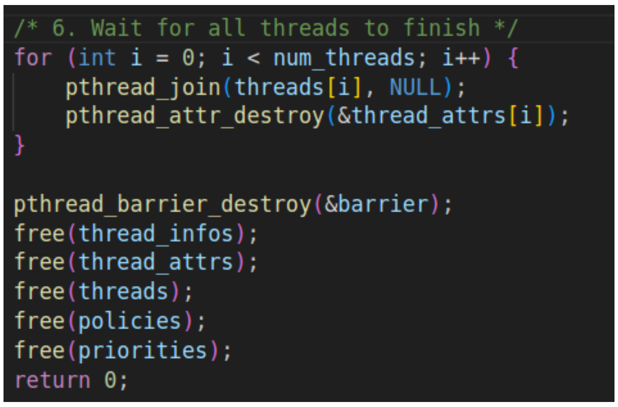
```c
// Create each thread: explicitly set policy/priority/affinity
for (int i = 0; i < num_threads; i++) {
    /* 4. Set attributes for each thread */
    thread_infos[i].thread_num = i;
    thread_infos[i].policy = policies[i];
    thread_infos[i].priority = priorities[i];
    thread_infos[i].busy_time = busy_time;

    pthread_attr_init(&thread_attrs[i]);
    pthread_attr_setaffinity_np(&thread_attrs[i], sizeof(set), &set);
    pthread_attr_setinheritsched(&thread_attrs[i], PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&thread_attrs[i], thread_infos[i].policy);

    struct sched_param sp = {0};
    sp.sched_priority = (thread_infos[i].policy == SCHED_FIFO) ? thread_infos[i].priority : 0;
    pthread_attr_setschedparam(&thread_attrs[i], &sp);

    pthread_create(&threads[i], &thread_attrs[i], thread_func, &thread_infos[i]);
}
```

g. main() - Start all threads at once
- 讓所有執行緒「幾乎同時」開始執行，避免建立先後造成偏差。



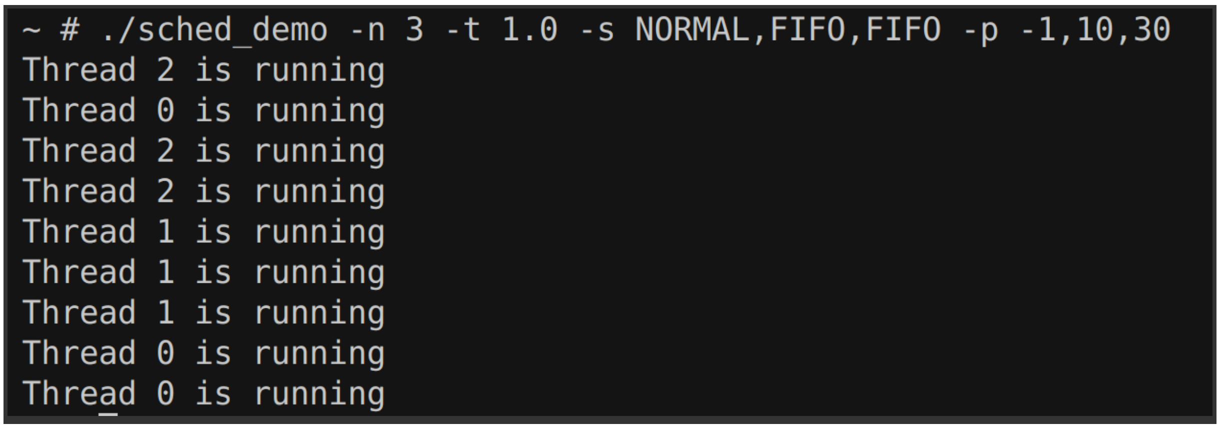
```c
/* 5. Start all threads at once */
pthread_barrier_wait(&barrier);
```

h. main() - Wait for all threads to finish
- 確保所有 worker threads 都完成，在釋放資源。



```c
/* 6. Wait for all threads to finish */
for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
    pthread_attr_destroy(&thread_attrs[i]);
}

pthread_barrier_destroy(&barrier);
free(thread_infos);
free(thread_attrs);
free(threads);
free(policies);
free(priorities);
return 0;
```

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

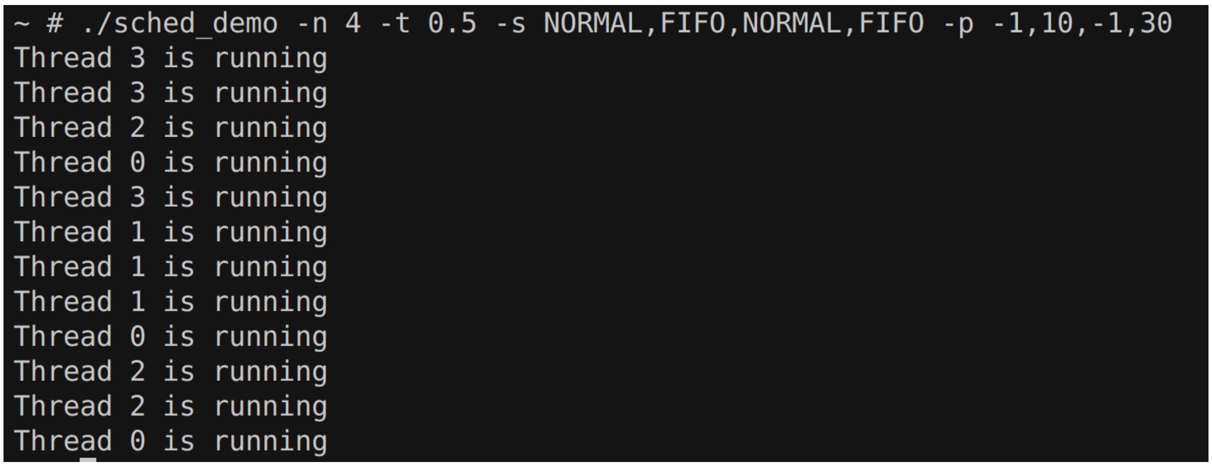


```
~ # ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 2 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 0 is running
Thread 0 is running
```

- Thread 2（FIFO, prio 30）→ 優先執行，幾乎壟斷 CPU。
- Thread 1（FIFO, prio 10）→ 僅在高優先權 FIFO 完成或讓出時執行。
- Thread 0（NORMAL）→ 最後執行，只有在沒有即時執行緒可用時才獲得 CPU。

一開始 Thread 1 與 Thread 2 同屬 FIFO 類，但 Thread 2 優先權較高（30 > 10），因此先獲得 CPU。Thread 1 必須等待 Thread 2 主動讓出 CPU 或執行完成後才有機會運行。Thread 0 屬於一般排程，僅能在所有 FIFO 執行緒完成或暫時讓出時執行。

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



```
~ # ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 3 is running
Thread 2 is running
Thread 0 is running
Thread 3 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 0 is running
Thread 2 is running
Thread 2 is running
Thread 0 is running
```

- Thread 3（FIFO, prio 30）→ 優先執行，幾乎壟斷 CPU
- Thread 2（NORMAL）→ 只有在沒有即時執行緒可用時才獲得 CPU
- Thread 1（FIFO, prio 10）→ 僅在高優先權 FIFO 完成或讓出時執行

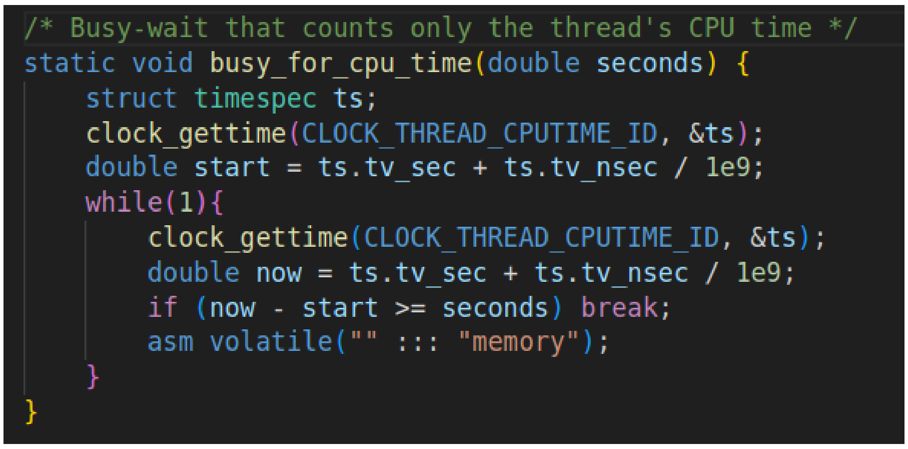
- Thread 0（NORMAL） → 只有在沒有即時執行緒可用時才獲得 CPU

高優先權的 FIFO 執行緒（Thread 3）會搶佔 CPU，最先執行且最頻繁。
低優先權的 FIFO 執行緒（Thread 1）會在高優先權執行緒讓出後執行。
NORMAL 執行緒（Thread 0 與 Thread 2） 只在沒有即時執行緒時運作

4. Describe how did you implement n-second-busy-waiting? (10%)

此函式會取得當前執行緒實際使用 CPU 的時間，當執行緒被排程器搶佔（例如被高優先權 FIFO 執行緒中斷）時，這個時間不會增加，只有當該執行緒真正執行在 CPU 上時，時間才會往前走。

若使用 sleep()，執行緒會進入等待狀態，由排程器重新排入 ready queue，這樣會使程式實際等待的時間包含「被搶佔的時間」，無法觀察不同排程政策（如 FIFO 與 NORMAL）對 CPU 時間的真實影響。



```
/* Busy-wait that counts only the thread's CPU time */
static void busy_for_cpu_time(double seconds) {
    struct timespec ts;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
    double start = ts.tv_sec + ts.tv_nsec / 1e9;
    while(1){
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
        double now = ts.tv_sec + ts.tv_nsec / 1e9;
        if (now - start >= seconds) break;
        asm volatile("" ::: "memory");
    }
}
```

5. What does the `kernel.sched_rt_runtime_us` effect? If this setting is changed (eg. 500000, 950000, 1000000), what will happen?(10%)

`kernel.sched_rt_runtime_us` 是 Linux kernel 中一個 Real-Time Scheduling 的資源限制參數，用來限制即時執行緒在整個 CPU 上能使用的時間比例。

設定值	意義	實際影響
`kernel.sched_rt_runtime_us = 500000`	即時執行緒最多佔用 50% CPU 時間	執行緒會被強制暫停，留出更多時間給一般執行緒；即時性降低，但系統更穩定
`kernel.sched_rt_runtime_us = 950000`	預設 95%	通常平衡了即時效能與系統反應，最常見設定
`kernel.sched_rt_runtime_us = 1000000`	允許 100%	即時執行緒可完全佔用 CPU（不再被限制）；若 FIFO 任務不釋放 CPU，普通執行緒

		可能完全無法執行