學號: 313551118　　姓名: 周哲瑋　　系級: 資科工碩一

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

第一步

Parse program arguments

使用 getopt 處理傳進來的參數

然後根據 "n:t:s:p:" 來 parse

```
int opt;
   while ((opt = getopt(argc, argv, "n:t:s:p:")) != -1) {
       switch (opt) {
           case 'n':
               num_threads = atoi(optarg);
               break;
           case 't':
               time_wait = atof(optarg);
               break;
           case 's':
               policy_str = strdup(optarg);
               break;
           case 'p':
               priority_str = strdup(optarg);
               break;
           default:
               fprintf(stderr, "Usage: %s -n <num_threads> -t
<time_wait> -s <policies> -p <priorities>\n", argv[0]);
               exit(EXIT_FAILURE);
       }
   }
```

判斷 thread 數量參數是否正確

一定要大於 0

但也要小於 MAX_THREADS, 這裡我設定成 30 (可以調整)。

目的是讓後面關於 thread 資訊的陣列先宣告長度為 MAX_THREADS。

當然也可以直接用 num_threads 宣告長度。

```
if (num_threads <= 0 || num_threads > MAX_THREADS) {
       fprintf(stderr, "Invalid number of threads.\n");
       exit(EXIT_FAILURE);
   }
```

再來是解析排程策略以及優先度
分別存到 policies 和 priorities 陣列。

```
// 解析排程策略
    char *token = strtok(policy_str, ",");
    for (int i = 0; i < num_threads && token != NULL; i++) {
        if (strcmp(token, "NORMAL") == 0) {
            policies[i] = SCHED_OTHER;
        } else if (strcmp(token, "FIFO") == 0) {
            policies[i] = SCHED_FIFO;
        } else {
            fprintf(stderr, "Invalid scheduling policy: %s\n", token);
            exit(EXIT_FAILURE);
        }
        token = strtok(NULL, ",");
    }

    // 解析優先級
    token = strtok(priority_str, ",");
    for (int i = 0; i < num_threads && token != NULL; i++) {
        priorities[i] = atoi(token);
        token = strtok(NULL, ",");
    }
```

第二步
Create <num_threads> worker threads
宣告 thread 以及之後會用到的資料以及屬性
另外 pthread_barrier_init 用來初始化 barrier。Barrier 是一種同步機制，它可以讓多個 thread 在指定的同步點上等待，直到所有 thread 都到達該點後才繼續執行。

```
// 2. 創建工作執行緒
    pthread_t threads[MAX_THREADS];
    pthread_attr_t attr;
    thread_data_t thread_data[MAX_THREADS];

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

下面的是自定義的 thread_data_t 的 struct,包含其 id,排程策略,優先度,等待時間

```
typedef struct {
   int thread_id;
   int policy;
   int priority;
   double time_wait;
} thread_data_t;
```

第三和四步
Set CPU affinity
Set the attributes to each thread
如註解所示, 前三行在設定 CPU 親和度 (affinity)。CPU_SET 我指定使用 cpu 0
接下來進入 for-loop 來創建 pthread , 其中每次都會設定 attr。
再來是設定執行緒屬性, 如排程政策和優先度, 其中 PTHREAD_EXPLICIT_SCHED 即 thread 使用 attr 中設定的排程策略和優先級, 而不繼承其創建者(父 thread)的排程屬性。
接下來把關於 thread 的資料放到 thread_data, 用以帶到 pthread_create 中。

```
// 3. 設置 CPU 親和性
   cpu_set_t cpuset;
   CPU_ZERO(&cpuset);
   CPU_SET(0, &cpuset);

   for (int i = 0; i < num_threads; i++) {
       pthread_attr_init(&attr);
       pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpuset);

       // 4. 設置執行緒屬性
       pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
       pthread_attr_setschedpolicy(&attr, policies[i]);

       struct sched_param param;
       param.sched_priority = priorities[i];
       pthread_attr_setschedparam(&attr, &param);

       thread_data[i].thread_id = i;
       thread_data[i].policy = policies[i];
       thread_data[i].priority = priorities[i];
       thread_data[i].time_wait = time_wait;

       int rc = pthread_create(&threads[i], &attr, thread_func, (void
*)&thread_data[i]);
```

```
        if (rc != 0) {
            fprintf(stderr, "Error creating thread %d: %s\n", i,
strerror(rc));
            exit(EXIT_FAILURE);
        }
        pthread_attr_destroy(&attr);
    }
```

thread_func 以及 busy_wait
等待所有執行緒同步啟動的 pthread_barrier_wait 我是放在這裡, 而不是放在 main 中。
這樣每個執行緒都會執行到(除了 main thread)。
接下來執行 3 次任務, 每次都會執行 busy_wait 。
busy_wait 使用 clock_gettime 來取得開始時間和當前時間, 當時間差值來到 time_wait 時, 就會結束。(詳細可見第4題的回答)

```
void busy_wait(double seconds) {
    struct timespec start, current;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start); //注意這裡使用的是
CLOCK_THREAD_CPUTIME_ID

    while (1) {
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &current);
        double elapsed = (current.tv_sec - start.tv_sec) +
                         (current.tv_nsec - start.tv_nsec) / 1e9;
        if (elapsed >= seconds) break;
    }
}

void *thread_func(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;

    // 等待所有執行緒同步啟動
    pthread_barrier_wait(&barrier);

    /* 2. 執行任務 */
    for (int i = 0; i < 3; i++) {
        printf("Thread %d is starting\n", data->thread_id);
        /* 忙碌 <time_wait> 秒 */
        busy_wait(data->time_wait);
    }


    pthread_exit(NULL);
}
```

第五步

Wait for all threads to finish

等待所有執行緒完成，然後釋放空間。

```c
// 5. 等待所有執行緒完成
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_barrier_destroy(&barrier);
    free(policy_str);
    free(priority_str);
```

運行測資結果

```
wayne@wayne-B450M-K:~/project/OS/lab2$ sudo ./sched_test.sh ./sched_demo ./sched_demo_313551118
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!
```

運行時間

```
wayne@wayne-MS-7D91:~/project/os/lab2$  sudo time  ./sched_demo_313551118 -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,1
0,-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 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
1.70user 4.29system 0:06.00elapsed 99%CPU (0avgtext+0avgdata 1920maxresident)k
0inputs+0outputs (0major+101minor)pagefaults 0swaps
```

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. (10%)**

執行題目的指令後，結果如下圖。

原因：

- FIFO 是 real-time policy,所以 thread 1 和 2 的權限較 normal 的 thread 0 高。而優先度 thread 2 (30) 又高於 thread 1 (10)。
- 因此 thread 2 總是會早於 thread 1，等 thread 2 全都執行完，才會執行 thread 1。而 thread 0 有一個出現在 thread 2 之間是因為預設 kernel.sched_rt_runtime_us=950000。所以 1秒中有 5% 必須讓給 normal thread，而 thread 0 剛好搶到這個 5% 來排程並執行。
- 順帶一題，在第五題我有講到 kernel.sched_rt_runtime_us 如果調的夠大，例如執行 sysctl -w kernel.sched_rt_runtime_us=1000000，則 normal 就必須得等所有 real-time threads 都執行完才結束。因此會出現 thread 2 連 3 次執行,然後 thread 1 連3次執行，最後 thread 0 連3次執行。

```
wayne@wayne-MS-7D91:~/project/os/lab2$ 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 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 0 is starting
```

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%)**

執行題目的指令後，結果如下圖。

原因:

- 同上一題的原因, real-time policy 的 FIFO (thread 3 和 thread 1) 有比 normal (thread 2 和 thread 0)更高的優先度。
- thread 3 的優先度(30) 又高於 thread 1 (10)，所以可以看到會先執行完 thread 3 再給 thread 1。
- 也和前一題相同，kernel.sched_rt_runtime_us=950000，所以執行 real-time thread 時，可能會偶有 normal thread 夾雜於之間執行。由於 normal thread 彼此優先度一樣，所以會 0->2 或 2->0 接力出來，但不會有一方連續3次執行。

```
wayne@wayne-MS-7D91:~/project/os/lab2$ 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
```

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

- thread 會在 printf 完後,執行 busy_wait(data->time_wait);
- busy_wait 函式會用 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start); 取得起始時間, 再來進入 while(1) 不斷循環, 每次都用 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &current);
- 取得當前時間, 和起始時間相減得到 elapsed, 也就是經過的時間。如果這個 elapsed 大於等於 time-wait (n-second) 則結束。
- 值得注意的是 clock_gettime 的參數不應該使用 CLOCK_MONOTONIC, 因為其他 thread 搶佔的時間也會被算入其中。所以用 CLOCK_THREAD_CPUTIME_ID 專門記入這個 thread 的執行時間, 當該 thread 被中斷或被其他 thread 搶佔時, 不會記錄這段時間。所以計時結果僅包含當前 thread 的執行時間, 而非實際的經過時間。

```
void busy_wait(double seconds) {
    struct timespec start, current;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start);
    while (1) {
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &current);
        double elapsed = (current.tv_sec - start.tv_sec) +
                         (current.tv_nsec - start.tv_nsec) / 1e9;
        if (elapsed >= seconds) break;
    }
}

void *thread_func(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    // 等待所有執行緒同步啟動
    pthread_barrier_wait(&barrier);

    /* 2. 執行任務 */
    for (int i = 0; i < 3; i++) {
        printf("Thread %d is starting\n", data->thread_id);
        /* 忙碌 <time_wait> 秒 */
        busy_wait(data->time_wait);
    }
    pthread_exit(NULL);
}
```

5. **What does the kernel.sched_rt_runtime_us effect? If this setting is changed, what will happen?(10%)**

- kernel.sched_rt_runtime_us 意思是一個時間段中所有實時(real-time)任務可以用的時間(以微秒為單位), 通常用來避免實時任務佔用過多的 CPU 資源, 影響到非實時任務的執行。
- 舉例來說, kernel.sched_rt_runtime_us = 950000, 表示一個時間段(預設1秒), 實時任務最多能執行 95% 的時間段。
- 超過這個時間後, 實時任務可能會被強制暫停, 讓出 CPU 給其他非實時任務。
- 如果這個參數設定過低, 實時任務的執行會更頻繁地被中斷, 可能導致延遲增加;相反, 設得太高則可能讓實時任務佔用過多資源, 影響整體系統的響應速度。
- 以第二題來說,如果我把 kernel.sched_rt_runtime_us 設的很大(1,000,000), 那麼執行的順序就不會有0跑到前面, 而是 2->2->2->1->1->1->0->0->0 , 因為 real-time threads 有更多的時間可以用, 造成低優先度的得等他全部執行完。
- 其結果可參考下圖(和第二題的圖的執行順序不同)。

```
test@test-Standard-PC-Q35-ICH9-2009:~/project/os/lab2$ sudo ./sched_demo_313551118 -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
```