LAB2 report

1.用 getopt 讀取 argc , argv 的參數，依照 n,t,s,p 來設定 thread 的 property，thread 的實作是依照讀進來的 thread 數來建立 thread vector，之後再用讀進來的 schedule policy 來設定 thread 的 attribute，也把 priority 指派給 thread，並把所有的 thread 都指派到同一個 CPU 後執行。

指派 CPU 的部分用 cpu_set_t 變數，priority 的部分則是用 sched_param 變數，policy 用 pthread_attr_t，所有都設定好後再用 pthread_create 建立 thread。

在 thread function 中，會先用一個 barrier 來擋住先進入 thread function 的 thread，直到所有的 thread 都建立完畢後，thread 才可以通過 barrier，並依照 priority、policy 來決定執行順序。

2.

```
Roy@os-next:~/os_lab2$ 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
```

會有這樣的原因是因為 real time thread 會先做，而數字越大 priority 越高，所以 thread2 會比 thread1 先，最後才會輪到 thread0

3.

```
Roy@os-next:~/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 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
```

Priority 高的先做，所以 thread3 比 thread1 先做，之後因為 thread0 跟 thread2 的 priority 相同，所以他們會輪流執行。

4.

```
for(int i = 0 ; i < 3 ; i++)
{
        std::cout << "Thread " << thread_id << " is starting" << std::endl;
        auto start_time = std::chrono::steady_clock::now();
        auto end_time = start_time + std::chrono::milliseconds(wait_time);
        //std::cout << wait_time << std::endl;
        while(std::chrono::steady_clock::now() < end_time)
        {
        //busy waiting
        }
}
```

計算 start_time 跟 end_time 後用 while 迴圈來達成 busy waiting
5.會規定 real time 的 thread 可以占 1 秒鐘的多少百分比，如果是 1000000，就代表 1 秒鐘 100%都執行 real time，如果是 950000，則代表 1 秒鐘 95%執行 real time，如果這樣設定，則在 real time thread 之間可能會有 normal thread 插進來執行。