Report

1. Describe how you implemented the program in detail. (20%)

First, we use **getopt()** to retrieve the arguments from the command line, including the number of threads, the wait time, the policies, and the priorities. Two custom functions, **parse_policies** and **parse_priorities**, are employed to parse the last two string arguments into arrays.

Second, we declare the thread-related data structures for later use. Additionally, we initialize a barrier using **pthread_barrier_init()**. This barrier will be used to synchronize the threads.

Next, to set the CPU affinity for all threads in the process, we utilize **sched_setaffinity()**.

Since we declared the array of attribute structures in the second step, we can now initialize the attribute structures using **pthread_attr_init()**, set them to the explicit scheduling policy with **pthread_attr_setinheritsched()**, and configure the corresponding policy and parameters. Note that only if the policy is FIFO should we set the priority.

After the attributes are set, we call **pthread_create()** to start the threads. The barrier declared in the second step can be placed at the start of the **thread_func()** for synchronization. We also include logic to check whether the CPU affinity is the same as what we set before. Then, the threads start their own busy waiting, as explained in question 4.

Returning to the main process, in the last step, we simply need to call **pthread_join()** for each thread and wait for them to finish. **pthread_attr_destroy()** and **pthread_barrier_destroy()** can also be employed to destroy the unused data structures.

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%)

As illustrated below, Thread 2 consistently executes before Thread 1 even begins. This occurs because we have assigned a higher priority to Thread 2 compared to Thread 1. Nevertheless, the execution order of Thread 0 remains unaffected by the other two threads, as it follows a distinct scheduling policy.

```
Case 3:
sudo ./sched_demo_311551137 -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 0 is running
Thread 2 is running
Thread 0 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
```

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%)

As depicted below, Thread 1 and 3 adhere to the FIFO policy, with the priority of Thread 3 set higher than that of Thread 1. Consequently, Thread 3 consistently executes before Thread 1 initiates. As for Thread 0 and 2, given their utilization of the NORMAL policy, CPU resources will be evenly distributed between them without a obvious order.

```
Case 4:
sudo ./sched_demo_311551137 -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 2 is running
Thread 0 is running
Thread 3 is running
Thread 2 is running
Thread 0 is running
Thread 3 is running
Thread 2 is running
Thread 0 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
```

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

I employ a do...while loop to continuously calculate the elapsed time until it reaches n seconds.