Operating System Homework Report 2

1. How to implemented the program in detail? 312706019

   - use comment to explain

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <sched.h>
#include <assert.h>

//for getopt (parse command line arguments)
int n = -1;
float t = -1.0;
char *s = NULL;
char *p = NULL;
//for barrier
pthread_barrier_t barrier;
//for mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//for busy waiting
double time_wait;
//stored thread info
typedef struct {
   int id;
   int policy;
   int priority;
} ThreadInfo;
//get the CPU time of the current thread in seconds function
static double my_clock(void) {
   struct timespec t;
   // ensure clock_gettime can excute successfully
   assert(clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) == 0);
   //calculate total seconds
   double seconds = t.tv_sec + 1e-9 * t.tv_nsec;
   return seconds;
}
//worker thread function
void *thread_func(void *arg) {
   /* 1. Wait until all threads are ready */
       pthread_barrier_wait(&barrier);
   /* 2. Do the task */
   for (int i = 0; i < 3; i++) {
```

```
      //the usage of a mutex to control critical section
       pthread_mutex_lock(&mutex);
       {
            printf("Thread %d is running\n", *((int *)arg));
            /* Busy for <time_wait> seconds */
               double sttime = my_clock();
               while (1) {
                   if (my_clock() - sttime > 1* time_wait)
                       break;
           }
       }
       pthread_mutex_unlock(&mutex);

   }
   /* 3. Exit the function */
   pthread_exit(NULL);
}

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

```
    // copy s(policies), p(priority) to policy_input, priority_input
    char *policy_input = strdup(s);
    char *priority_input = strdup(p);
    char policy_array[n][10];
    char priority_array[n][10];
    int policy[n];
    int priority[n];

    int id = 0;
    // copy policy and priority to  policy_array and priority_array, and ","
is the delimiter
    char *policy_tokens = strtok(policy_input, ",");
    strcpy(policy_array[id++], policy_tokens);

    while ((policy_tokens = strtok(NULL, ",")) != NULL) {
        strcpy(policy_array[id++], policy_tokens);
    }

    id = 0;
    char *priority_tokens = strtok(priority_input, ",");
    strcpy(priority_array[id++], priority_tokens);

    while ((priority_tokens = strtok(NULL, ",")) != NULL) {
        strcpy(priority_array[id++], priority_tokens);
    }

    free(policy_input);
    free(priority_input);

    //set policy's value for SCH_OTHER, SCH_FIFO
    //set priority's value to integer
    for (int i = 0; i < n; i++) {
        if (strcmp(policy_array[i], "NORMAL") == 0) {
            policy[i] = SCHED_OTHER;
        } else if (strcmp(policy_array[i], "FIFO") == 0) {
            policy[i] = SCHED_FIFO;
        } else {
            fprintf(stderr, "Error: Unknown scheduling policy.\n");
         exit(EXIT_FAILURE);
        }
        priority[i] = atoi(priority_array[i]);
    }

    /* 2. Create <num_threads> worker threads */
```

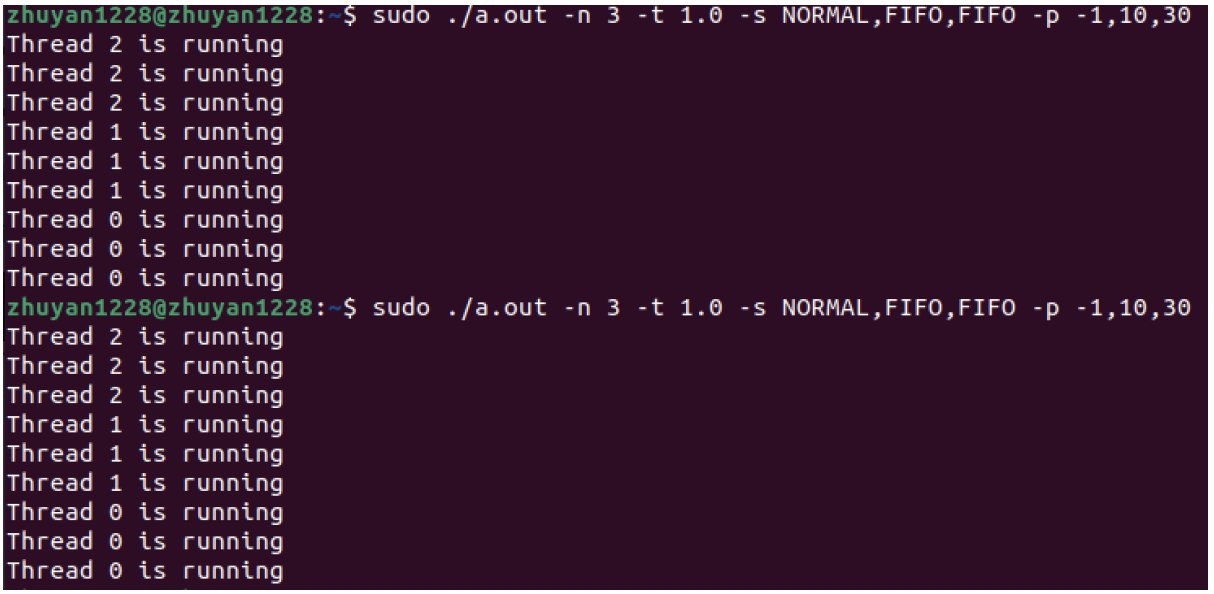
```
 //create thread id and thread info
   //malloc memory for tid and threadInfo
   //set time_wait
   pthread_t *tid = (pthread_t *)malloc(sizeof(pthread_t) * n);
   ThreadInfo *threadInfo = (ThreadInfo *)malloc(sizeof(ThreadInfo) * n);
   time_wait = t;

   //set threadInfo's value for n thread
   for (long i = 0; i < n; i++) {
       threadInfo[i].id = i;
       threadInfo[i].policy = policy[i];
       threadInfo[i].priority = priority[i];
   }

   /* 3. Set CPU affinity */
   //the process will be executed on CPU 0
   cpu_set_t set; CPU_ZERO(&set); CPU_SET(0, &set);
   //set CPU affinity for the thread which is calling
   sched_setaffinity(getpid(), sizeof(set), &set);

   /* 4. Set the attributes to each thread */
   //set attr
   pthread_attr_t attr;

   //initial barrier
   pthread_barrier_init(&barrier, NULL, n + 1);
   //create n thread
   for (long i = 0; i < n; i++) {
       //initial attr
       pthread_attr_init(&attr);
       //set inheritsched to explicit, ensure the thread's scheduling policy
and priority will be set by the attr we set
       pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
       //set scheduling policy and priority
       pthread_attr_setschedpolicy(&attr, threadInfo[i].policy);
       struct sched_param param;
       param.sched_priority = threadInfo[i].priority;
       pthread_attr_setschedparam(&attr, &param);
       //create thread, and pass thread id and attr to thread_func
       int result = pthread_create(&tid[i], &attr, thread_func, (void
*)&threadInfo[i].id);
       if (result != 0) {
           fprintf(stderr, "Error creating thread %ld: %s\n", i,
strerror(result));
           exit(EXIT_FAILURE);
```

```
        }
    }
    //wait until all threads are ready
    pthread_barrier_wait(&barrier);

    /* 5. Start all threads at once */
    for (int i = 0; i < n; i++) {
        //join thread
        int result = pthread_join(tid[i], NULL);
        if (result != 0) {
            fprintf(stderr, "Error joining thread %d: %s\n", i,
strerror(result));
            exit(EXIT_FAILURE);
        }
    }
    /* 6. Wait for all threads to finish  */
    //destroy barrier
    pthread_barrier_destroy(&barrier);
    free(tid);
    free(threadInfo);

    return 0;
}
```

This is the result of the test cases.

```
zhuyan1228@zhuyan1228:~$ sudo ./sched_test.sh ./sched_demo ./a.out
Running testcase 1: ./sched_demo -n 1 -t 0.5 -s NORMAL -p -1 ......
Result: Success!
Running testcase 2: ./sched_demo -n 2 -t 0.5 -s FIFO,FIFO -p 10,20 ......
Result: Success!
Running testcase 3: ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30 ...
...
Result: Success!
```

2. Describe the results of ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30 and what causes that:
   - Threads 1 and 2 use FIFO policy, so they have real-time policy priority and the bigger priority value has higher priority; Thread 0 uses Normal policy so it doesn't have real-time policy priority(-1), assigned lower priorities as it is a standard scheduling policy.
   - Use setaffinity let them run on the same cpu.
   - When a SCHED_FIFO thread becomes runnable, it will always immediately preempt any currently running thread.
   - In that case, most likely, thread 2 will be run first (priority(30) > thread 1(10) > thread 0(-)), and thread 0 will be run at the end.
   - Thread 2 -> Thread1 -> Thread 0



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:
   - Thread 3 and thread 1 use FIFO policy, and the others use Normal policy, so they may run in front of them.
   - Thread 3 have a higher priority than thread 1, it will run before thread 1 because it FIFO policy.
   - Use setaffinity let them run on the same cpu.
   - Normal (SCHED_OTHER) policy: assign a time slice to the thread during which it can execute. It's possible that thread 0 and 2 will fight for using CPU. May cause preempt.
   - Thread3 -> Thread 1 -> Thread 0 or 2

```
zhuyan1228@zhuyan1228:~$ sudo ./a.out -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 3 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 0 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
```

```
zhuyan1228@zhuyan1228:~$ sudo ./a.out -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 3 is running
Thread 3 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
```

4. Describe how did you implement n-second-busy-waiting?
   - **`my_clock`** This function gets the current thread's CPU time in seconds, used for measuring time.

```c
//get the CPU time of the current thread in seconds function
static double my_clock(void) {
    struct timespec t;
    // ensure clock_gettime can excute successfully
    assert(clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) == 0);
    //calculate total seconds
    double seconds = t.tv_sec + 1e-9 * t.tv_nsec;
    return seconds;
}
```

   - The busy-wait loop simulates some computation or delay in each thread

- In thread_func, the busy-wait loop in the section doing the task, enters a busy-wait loop for time_wait(t) seconds using the my_clock function, ensure n-second-busy-waiting.

```
/* Busy for <time_wait> seconds */
               double sttime = my_clock();
               while (1) {
                   if (my_clock() - sttime > 1* time_wait)
                       break;
```

- Thread_func

```
//worker thread function
void *thread_func(void *arg) {
   /* 1. Wait until all threads are ready */
       pthread_barrier_wait(&barrier);
   /* 2. Do the task */
   for (int i = 0; i < 3; i++) {
       //the usage of a mutex to control critical section
       pthread_mutex_lock(&mutex);
       {
            printf("Thread %d is running\n", *((int *)arg));
            /* Busy for <time_wait> seconds */
               double sttime = my_clock();
               while (1) {
                   if (my_clock() - sttime > 1* time_wait)
                       break;
           }
       }
       pthread_mutex_unlock(&mutex);

   }
   /* 3. Exit the function */
   pthread_exit(NULL);
}
```