

2024-Fall Operating System Assignment II

This page demonstrates my answer to the quiz in HW2, for further detail of this assignment, please check [this link](#)

- ◆ Student Name : 許登豪
- ◆ Student ID : 313551026

Q1: Describe how you implemented the program in detail.

 Functions that parse the input command-line options :

```
void parse_args(int argc, char* argv[], int* number_of_threads, double* time_, char* policy_string, char* priority_string){
    int opt;
    while((opt = getopt(argc, argv, "n:t:s:p:")) != -1){
        switch(opt){
            case 'n':
                *number_of_threads = atoi(optarg);
                break;
            case 't':
                *time_ = atof(optarg);
                break;
            case 's':
                strcpy(policy_string, optarg);
                break;
            case 'p':
                strcpy(priority_string, optarg);
                break;
            default:
                fprintf(stderr, "Usage: %s [-n number] [-t time] [-s scheduling_policies] [-p priorities]\n", argv[0]);
                exit(EXIT_FAILURE);
        }
    }
}
```

```
char* parse_per_thread_info(int* i, char* str){
    char* per_thread_info = (char*)malloc(100*sizeof(char));
    int cur = 0;
    while(str[*i] != ',' && str[*i] != '\0'){
        per_thread_info[cur++] = str[*i];
        *i += 1;
    }
    per_thread_info[cur] = '\0';
    return per_thread_info;
}
```

 Brief explanation of the above 2 functions :

The `parse_args()` function uses `getopt()` to parse the command-line options `(-n, -t,`

`-s, -p`) of all threads and use `atoi()` and `atof()` to convert them to correct data type.

Scheduling policies and priorities of all threads are passed as comma-separated strings `(policy_string, priority_string)` and are parsed into substring individually for each thread in `parse_per_thread_info()`.

📷 Functions that set the cpu affinity of all threads :

```
void set_cpu_set(cpu_set_t* cpuset){
    CPU_ZERO(cpuset);
    CPU_SET(0, cpuset);
}
```

✍️ Brief explanation of the above function :

The `set_cpu_set()` function receives a reference of `cpu_set_t*` variable and sets this variable so that all threads can only run on CPU 0.

📷 Data structures storing each thread's information and the shared variable :

```
pthread_barrier_t barrier; // threads share global variable


typedef struct {
    pthread_t thread_id;
    double time_wait;
    int thread_num;
    int sched_policy;
    int sched_priority;
} thread_info_t;
```

✍️ Brief explanation of the above structure and the barrier :

In the `thread_info_t` structure, the `thread_id` stores the thread id of this thread ; `time_wait` stores how long this threads needs to busy-waiting in seconds ; `thread_num` represents the number of this thread that is going to print on the screen (differnt than `thread_id`) ; `sched_policy` and `sched_priority` are the scheduling policy and the priority of this thread in `int` type, because in pthread library policy-related macros like `SCHED_NORMAL`, `SCHED_FIFO` are defined as `int` type.

For the `pthread_barrier_t barrier` part, it is used to "block" all threads to wait "until all threads are ready to run".

Otherwise the order that threads run will be non-deterministic because we can not know in advance when OS will make each thread active.

 Main Function that does all the work

```
int main(int argc, char *argv[]){
    int number_of_threads;
    double time_;
    char* policy_string = (char*)malloc(1000*sizeof(char));
    char* priority_string = (char*)malloc(1000*sizeof(char));
    parse_args(argc, argv, &number_of_threads, &time_, policy_string, priority_string);
    int index1 = 0;
    int index2 = 0;

    cpu_set_t cpuset;
    set_cpu_set(&cpuset);

    pthread_barrier_init(&barrier, NULL, number_of_threads);
    thread_info_t* tinfo;
    tinfo = (thread_info_t*)malloc(number_of_threads * sizeof(thread_info_t));
    char *policy, *priority;
```

```

for(int i = 0; i < number_of_threads; i++){
    tinfo[i].thread_num = i;
    tinfo[i].time_wait = time_;
    policy = parse_per_thread_info(&index1, policy_string);
    priority = parse_per_thread_info(&index2, priority_string);
    index1++;
    index2++;

    if(strcmp(policy, "NORMAL") == 0){
        tinfo[i].sched_policy = SCHED_OTHER;
    }
    else if (strcmp(policy, "FIFO") == 0) {
        tinfo[i].sched_policy = SCHED_FIFO;
    }
    else{
        tinfo[i].sched_policy = SCHED_RR;
    }
    tinfo[i].sched_priority = atoi(priority) < 0 ? 0 : atoi(priority);

    pthread_attr_t t_attr;
    struct sched_param schedParam;
    pthread_attr_init(&t_attr);
    pthread_attr_setschedpolicy(&t_attr, tinfo[i].sched_policy);
    schedParam.sched_priority = tinfo[i].sched_priority;
    pthread_attr_setschedparam(&t_attr, &schedParam);
    pthread_attr_setaffinity_np(&t_attr, sizeof(cpuset), &cpuset);
    pthread_attr_setinheritsched(&t_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_create(&(tinfo[i].thread_id), &t_attr, thread_func, &tinfo[i]);
    pthread_attr_destroy(&t_attr);
}

```

```

for(int i = 0; i < number_of_threads; i++){
    pthread_join(tinfo[i].thread_id, NULL);
}

pthread_barrier_destroy(&barrier);
free(policy_string);
free(priority_string);
free(tinfo);
return 0;
}

```

 Brief explanation of the main function :

First, parse all threads argument using `parse_args()` .

Second, set the cpu affinity using `set_cpu_set()` .

Third, initialize the thread barrier using `pthread_barrier_init()` to synchronize all thread.

Fourth, malloc a chain of memory `tinfo` to store per thread's information

(`thread_info_t`)

Next, every iteration in the for loop will

a. parse its scheduling policy and priority using `parse_per_thread_info()` and set the


data field of each thread's `thread_info_t`.

b. initialize a `pthread_attr_t` `t_attr`, which we can think of it as a container to store the thread's attribute, and then we can assign this "container" to our target thread.

c. use `pthread_attr_setschedpolicy(&t_attr, tinfo[i].sched_policy)` to set the scheduling policy information in `t_attr`; use `pthread_attr_setschedparam(&t_attr, &schedParam)` to set the priority in `t_attr`; use `pthread_attr_setaffinity_np(&t_attr, sizeof(cpu_set_t), &cpu_set)` to set the cpu affinity in `t_attr`; use `pthread_attr_setinheritsched(&t_attr, PTHREAD_EXPLICIT_SCHED)` to tell the kernel that we don't want our worker thread inherits the attribute of the main thread, which is a default behavior if not specified.

d. create the thread with our specific attribute `t_attr` using `pthread_create()`, with each thread doing the work defined in `thread_func()`.

Last, join all the worker threads using `pthread_join()` to wait for all worker threads get their job done and then we move on to destroy barrier and free in-use memory in main thread and then terminate the main thread.

 The work each thread needs to do :

```
void *thread_func(void *arg){
    thread_info_t* tinfo_i;
    tinfo_i = (thread_info_t*)arg;
    pthread_barrier_wait(&barrier);
    int busy_time_msec = tinfo_i->time_wait * 1000;
    for (int i = 0; i < 3; i++) {
        printf("Thread %d is starting\n", tinfo_i->thread_num);
        /* Busy for <time_wait> seconds */
        struct timespec start_time, end_time;
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start_time);
        int start_time_msec = start_time.tv_sec * 1000 + start_time.tv_nsec / 1000000;
        while(1){
            clock_gettime(CLOCK_THREAD_CPUTIME_ID, &end_time);
            int end_time_msec = end_time.tv_sec * 1000 + end_time.tv_nsec / 1000000;
            if ((end_time_msec - start_time_msec) >= busy_time_msec) {
                break;
            }
        }
        sched_yield();
    }
    pthread_exit(NULL);
}
```

 Brief explanation of the above function :

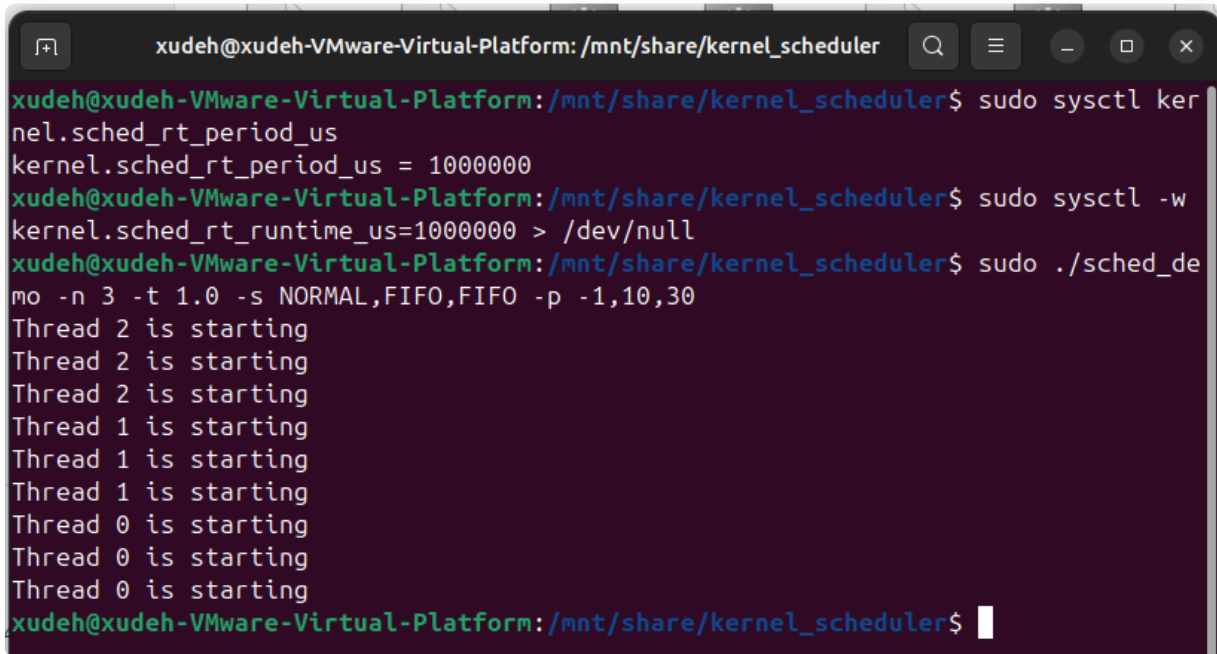
Each thread first turn the input argument `args` of type `void*` into type `thread_info_t*` so that they can read the data field correctly.

And then threads wait for others on the barrier until all threads are ready to run.

Then the workload is simply what is defined in the hackmd : loop 3 times and each iteration we need to busy waiting for `time_wait` seconds in order to mimic that this thread gets the CPU runtime and is doing something.

📌 **Q2: Describe the results of `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` and what causes that.**

🖼️ Screenshot of running `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` :

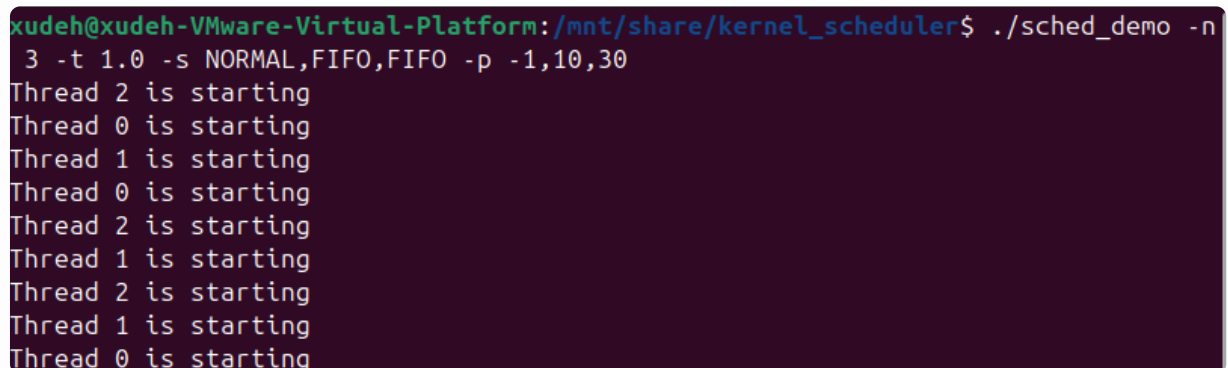


```
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ sudo sysctl kernel.sched_rt_period_us
kernel.sched_rt_period_us = 1000000
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ sudo sysctl -w kernel.sched_rt_runtime_us=1000000 > /dev/null
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ 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
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$
```

🔧 Why it is the case after setting `sched_rt_period_us` and `sched_rt_runtime_us` :

Since `SCHED_FIFO` is a real-time scheduling policy and has priority over the non-real-time `SCHED_NORMAL` , the execution order will be thread 2 (with the higher priority of 30 over 10), followed by thread 1, and finally the `SCHED_NORMAL` thread 0.

🖼️ Screenshot of running `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` :





```
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Thread 2 is starting
Thread 0 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 1 is starting
Thread 2 is starting
Thread 1 is starting
Thread 0 is starting
```

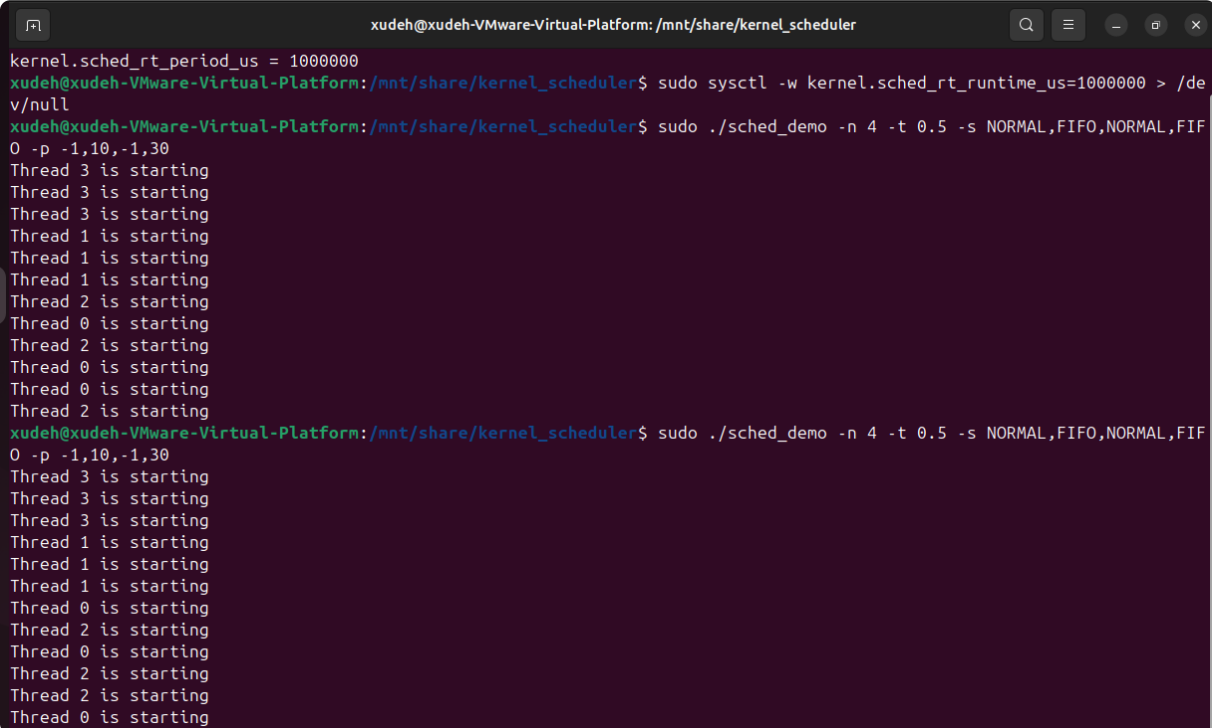
🔧 Why it is the case without setting `sched_rt_period_us` and `sched_rt_runtime_us` :

When all threads reach the barrier, thread 2 will execute first because it is the highest-priority real-time thread. However, since we did not allocate the entire scheduling period to the real-time thread (default is 95% for real time thread, 5% for non-real time thread), we will see thread 0 interleaving between the real-time thread 2 and thread 1 to fill the system's reserved execution time for non-real-time threads.


Nevertheless, the overall completion order remains reasonable: the highest-priority real-time thread 2 completes its three loops first, followed by the next highest-priority real-time thread 1, and finally, the non-real-time thread 0.

 **Q3: 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.**


 Screenshot of running `/sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30` :



```
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler
kernel.sched_rt_period_us = 1000000
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ sudo sysctl -w kernel.sched_rt_runtime_us=1000000 > /dev/null
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
0 -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 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 0 is starting
Thread 2 is starting
xudeh@xudeh-VMware-Virtual-Platform: /mnt/share/kernel_scheduler$ sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
0 -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 2 is starting
Thread 2 is starting
Thread 0 is starting
```


 Why it is the case after setting `sched_rt_period_us` and `sched_rt_runtime_us` :

Since `SCHED_FIFO` (threads 1 and 3) is a real-time policy, it has higher priority than `SCHED_NORMAL` (threads 0 and 2). Additionally, thread 3 has a higher priority than thread 1 (with the higher priority of 30 over 10), so the execution order is thread 3 first, followed by thread 1. Finally, threads 0 and 2, which have no priority advantage, will interleave since CFS scheduling in Linux comes to work.

 Screenshot of running `/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 2 is starting
Thread 0 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
```

 Why it is the case without setting `sched_rt_period_us` and `sched_rt_runtime_us` :

First 4 lines:

When all threads reach the barrier, thread 3 will execute first because it is the highest-priority real-time thread. In this example, after completing its three loops, there is still remaining time allocated for real-time threads. Therefore, thread 1, which is also a real-time thread, will continue execution.


Lines 4 to 8:


When the time allocated for real-time threads is exhausted, we will see thread 0 and thread 2 interleaving within the execution of real-time thread 1 (since thread 3 has already completed its three loops and will no longer appear) to fill the system's reserved execution time for non-real-time threads.

Lines 9 to the end:

After the last real-time thread (thread 1) finishes, the two non-real-time threads, thread 0 and thread 2, which have no priority differentiation, will take turns executing based on their respective runtime. This interleaving behavior is due to Linux's Completely Fair Scheduler (CFS), which strives to ensure equal execution time for all threads.

However, the overall completion order remains logical: the highest-priority real-time thread 3 completes its three loops first, followed by the next highest-priority real-time thread 1, and finally, the non-real-time threads 0 and 2 interleave execution due to CFS.

 **Q4: Describe how did you implement n-second-busy-waiting?**

 Screenshot of my busy waiting part :

```
24     int busy_time_msec = tinfo_i->time_wait * 1000;
25     for (int i = 0; i < 3; i++) {
26         printf("Thread %d is starting\n", tinfo_i->thread_num);
27         /* Busy for <time_wait> seconds */
28         struct timespec start_time, end_time;
29         clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start_time);
30         int start_time_msec = start_time.tv_sec * 1000 + start_time.tv_nsec / 1000000;
31         while(1){
32             clock_gettime(CLOCK_THREAD_CPUTIME_ID, &end_time);
33             int end_time_msec = end_time.tv_sec * 1000 + end_time.tv_nsec / 1000000;
34             if ((end_time_msec - start_time_msec) >= busy_time_msec) {
35                 break;
36             }
37         }
38         sched_yield();
39     }
```

 How I implement n-second-busy-waiting :


I use `clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start_time)` to get the current time and store it in the `start_time` variable at the beginning of the busy wait, using the `CLOCK_THREAD_CPUTIME_ID` clock, which retrieves the CPU time consumed by "only the calling thread".

That is, this `CLOCK_THREAD_CPUTIME_ID` clock only advances when the calling thread is actively using the CPU, which is what we want.

`while (1) { ... }` is a loop that continues as long as the `end_time_msec - start_time_msec >= busy_time_msec`, which means this thread have been busy-looping in this while loop for equal to or more than the time we are given.

When we leave the while loop, we need to voluntarily yeild the CPU using `sched_yield()` so that we do not consume CPU time more than we expected.

Q5: What does the `kernel.sched_rt_runtime_us` effect? If this setting is changed, what will happen? (10%)


 What does the `kernel.sched_rt_runtime_us` effect :

`kernel.sched_rt_runtime_us` defines the maximum amount of time (in microseconds) that real-time tasks are allowed to consume within each scheduling period, defined by `kernel.sched_rt_period_us`.

Setting `kernel.sched_rt_runtime_us = -1` will disable the real-time runtime limit entirely.

The `kernel.sched_rt_runtime_us` setting in Linux controls the time budget allowed for real-time tasks in the kernel's scheduling system. It limits the total CPU time available to

real-time tasks over a defined period to ensure that non-real-time tasks also get a chance to run.

 What will happen if this setting is changed :

a. Increasing `kernel.sched_rt_runtime_us` :

Effect: Real-time tasks get a larger share of CPU time within each period.

Pros :

Better performance for real-time tasks such as tasks requiring low-latency and predictable timing.

We may have a better chance of predicting the possible execution order in a system where real-time and non-real-time processes are combined.

Cons :

Risk of starvation. If `kernel.sched_rt_runtime_us` is set too close to `kernel.sched_rt_period_us` , non-real-time tasks might not get sufficient CPU time, leading to starvation or sluggish performance for general applications and background tasks.

b. Decreasing `kernel.sched_rt_runtime_us` :

And vice versa.