Operating System Fall 2024 Assignment #2

Student ID: 313551099
Student Name: 李以恩

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

1. Parse the arguments

- Use getopt(3) to parse the arguments
- The meaning of the command-line arguments:
    - -n : Number of threads to run simultaneously
    - -t : Duration of “busy” period
    - -s: Scheduling policy for each thread, SCHED_FIFO or SCHED_NORMAL
    - -p: Real-time thread priority for real-time threads

```
void parse_arguments(int argc, char *argv[])
{
    int opt;
    char *str_policy, *str_priority;
    while((opt = getopt(argc, argv, "t:n:s:p:")) != -1)
    {
        // printf("opt = %c\n", opt);
        // printf("optarg = %s\n", optarg);
        switch(opt)
        {
            case 'n':
                num_threads = atoi(optarg);
                break;
            case 't':
                time_wait = atof(optarg);
                break;
            case 's':
                str_policy = optarg;
                for(char *token=strtok(str_policy, ","); token!=NULL; token=strtok(NULL, ","))
                    policies.push_back(token);
                break;
            case 'p':
                str_priority = optarg;
                for(char *token=strtok(str_priority, ","); token!=NULL; token=strtok(NULL, ","))
                    priorities.push_back(atoi(token));
                break;
            default:
                cerr << "Usage: " << argv[0] << " -n <num_threads> -t <time_wait> -s <policies> -p <priorities>\n";
                exit(EXIT_FAILURE);
        }
    }
}
```

2. Create <num_threads> worker threads

- Use Thread_info structure to store the information for each thread, like thread id, schedule policy, priority.

```
/* the structure to store thread */
struct Thread_info
{
    int id;
    string policy;
    int priority;
};
```

- Use vectors to store the information of all the threads.

```
/* 2. Create <num_threads> worker threads */
vector<pthread_t> threads(num_threads);
vector<Thread_info> threads_infos(num_threads);
for(int i=0; i<num_threads; i++)
{
    threads_infos[i] = {i, policies[i], priorities[i]};
}
```

3. Set CPU affinity

- Use CPU_ZERO and CPU_SET to set CPU affinity, assigning all threads to CPU 0.

```
/* 3. Set CPU affinity */
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(0, &cpuset);
```

4. Set the attributes to each thread

- Create thread attributes with pthread_attr_t and configure them using:
  - pthread_attr_setaffinity_np to assign the CPU.
  - pthread_attr_setschedpolicy to set the scheduling policy.
  - pthread_attr_setschedparam to set the priority.
- Use pthread_create to create threads with the configured attributes.
- After creating threads, use pthread_attr_destroy to clean up the attributes.

```
/* 4. Set the attributes to each thread */
for(int i=0; i<num_threads; i++)
{
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpuset);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    sched_param param;
    param.sched_priority = threads_infos[i].priority;
    if(threads_infos[i].policy == "FIFO")
    {
        pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
        pthread_attr_setschedparam(&attr, &param);
    }
    else if(threads_infos[i].policy == "NORMAL")
    {
        pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    }
    if (pthread_create(&threads[i], &attr, thread_function, &threads_infos[i]) != 0)
    {
        cerr << "Error: Unable to create thread " << i << "\n";
        pthread_attr_destroy(&attr);
        return 1;
    }
    pthread_attr_destroy(&attr);
```

5. Start all threads at once

- Use a barrier to ensure all threads start at the same point:
  - Declare and initialize the barrier.
  - In the thread function, use pthread_barrier_wait to synchronize threads at the same starting point.
  - Destroy the barrier after use to clean up resources.

```
/* 5. Start all threads at once */
if (pthread_barrier_init(&barrier, NULL, num_threads) != 0)
{
    perror("pthread_barrier_init");
    return 1;
}
```

6. Wait for all threads to finish

- Use pthread_join to wait for all threads to complete before proceeding with the rest of the program.

```
/* 6. Wait for all threads to finish  */
for (int i = 0; i < num_threads; i++)
{
    pthread_join(threads[i], nullptr);
}
pthread_barrier_destroy(&barrier);
```

7. Inside the thread function

- Use pthread_barrier_wait to synchronize threads at the same starting point.
- Print out the information and do the busy wait.
- Exit the function.

```
void *thread_function(void *arg)
{
    Thread_info *info = (Thread_info *)arg;

    /* 1. Wait until all threads are ready */
    pthread_barrier_wait(&barrier);

    /* 2. Do the task */
    for(int i=0; i<3; i++)
    {
        printf("Thread %d is starting\n", info->id);
        busy_wait(time_wait);
    }

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

II. 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%)

id	sched_policy	sched_priority
0	Normal	-1
1	FIFO	10
2	FIFO	30

Thread2, with the highest priority (priority 30) and using the FIFO scheduling policy, is executed first. This is followed by thread1 (priority 10) and then thread0 (priority -1), which completes last.

```
ieen@Ubuntu:~/Desktop/OS_hw2$ 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
```

III. 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%)

id	sched_policy	sched_priority
0	Normal	-1
1	FIFO	10
2	Normal	-1
3	FIFO	30

Threads 3 (priority 30) and 1 (priority 10) adopt FIFO and have higher priority and will run first. Afterward, thread 0 and thread 2, using the Completely Fair Scheduler (CFS), will execute with equal fairness.

```
ieen@Ubuntu:~/Desktop/OS_hw2$ 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 2 is starting
Thread 0 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
```

IV. Describe how you implement n-second-busy-waiting. (10%)

- To simulate a specified busy-wait time, threads must perform intensive computations instead of sleeping (`sleep()` or `nanosleep()`).
- Use infinite loop (while(1)) to occupy CPU time.
- The clock_gettime() function is used to measure elapsed time until the specified duration (time_wait) is reached.

```c
void busy_wait(double time_wait)
{
    struct timespec start_time, current_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    while (true)
    {
        clock_gettime(CLOCK_MONOTONIC, &current_time);
        double elapsed_time = (current_time.tv_sec - start_time.tv_sec) +
                              (current_time.tv_nsec - start_time.tv_nsec) / 1e9;
        if (elapsed_time >= time_wait)
            break;
    }
}
```

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

- kernel.sched_rt_runtime_us specifies the maximum CPU time real-time threads can use within each scheduling period.
- With the default settings of sched_rt_period_us = 1,000,000 and sched_rt_runtime_us = 950,000, real-time threads can use up to 95% of the CPU time.
- Increasing sched_rt_runtime_us might allow real-time threads to monopolize the CPU, while decreasing it could impact real-time performance.

- Below is the experiment I did by changing kernel.sched_rt_runtime_us:

1. Before changing:

   kernel.sched_rt_runtime_us = 950000

```
ieen@Ubuntu:~/Desktop/OS_hw2$ sudo sysctl -w kernel.sched_rt_runtime_us=950000
kernel.sched_rt_runtime_us = 950000
ieen@Ubuntu:~/Desktop/OS_hw2$ 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 0 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 0 is starting
```

   Since real-time processes have higher priority than normal processes, the expected thread execution order is Thread 2 → Thread 1 → Thread 0. However, if sched_rt_runtime_us is smaller than the execution time required by real-time tasks, the real-time tasks will be constrained, allowing normal tasks to interleave. As a result, parts of Thread 0 may execute between the real-time processes.

2. After changing:

   kernel.sched_rt_runtime_us = 1000000 (= sched_rt_period_us)

```
ieen@Ubuntu:~/Desktop/OS_hw2$ sudo sysctl -w kernel.sched_rt_runtime_us=1000000
kernel.sched_rt_runtime_us = 1000000
ieen@Ubuntu:~/Desktop/OS_hw2$ 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
```