NYCU Operation System Homework 2

IOC PhD 徐浩哲 411551005

1. Describe how you implemented the program in detail.

- Parsing Program Arguments

  This task involves extracting and interpreting command-line arguments provided to the program, such as the number of threads, waiting time, scheduling policies, and priorities.

```c
void parse_arguments(int argc, char *argv[], int *num_threads, double
*time_wait, int **policies, int **priorities) {
int opt;
char *schedulers_csv = NULL;
char *priorities_csv = NULL;

while ((opt = getopt(argc, argv, "n:t:s:p:")) != -1) {
    switch (opt) {
    case 'n':
        *num_threads = atoi(optarg);
        break;
    case 't':
        *time_wait = atof(optarg);
        break;
    case 's':
        schedulers_csv = optarg;
        break;
    case 'p':
        priorities_csv = optarg;
        break;
    default: /* '?' */
        fprintf(stderr, "Usage: %s -n num_threads -t time_wait -s policies
p priorities\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}

// Allocate memory for policies and priorities arrays
*policies = malloc(*num_threads * sizeof(int));
*priorities = malloc(*num_threads * sizeof(int));

// Parse the schedulers and priorities
char *token;
int index = 0;

// Parse schedulers
token = strtok(schedulers_csv, ",");
while (token != NULL && index < *num_threads) {
    if (strcmp(token, "NORMAL") == 0) {
        (*policies)[index] = SCHED_OTHER;
    } else if (strcmp(token, "FIFO") == 0) {
        (*policies)[index] = SCHED_FIFO;
    } else {
        fprintf(stderr, "Invalid scheduler policy: %s\n", token);
        exit(EXIT_FAILURE);
    }
    token = strtok(NULL, ",");
    index++;
}

// Parse priorities
token = strtok(priorities_csv, ",");
index = 0;
while (token != NULL && index < *num_threads) {
    (*priorities)[index] = atoi(token);
    token = strtok(NULL, ",");
    index++;
}
```

- Create worker threads

  In this task, the code creates multiple threads (referred to as "worker threads") to perform concurrent tasks. The number and characteristics of these threads are determined based on the program's configuration.

```
for (int i = num_threads - 1; i >= 0; i--)  {
    // Initialize thread attribute
    pthread_attr_init(&attr);

    //printf("policies[%d] = %d priority = %d \n", i, policies[i], priorities[i]);

    // Set the scheduling policy in the attribute structure
    pthread_attr_setschedpolicy(&attr, policies[i]);

    // Set the scheduling priority in the attribute structure only if it's not SCHED_NORMAL
    if (policies[i] != SCHED_OTHER) {
        param.sched_priority = priorities[i];
        pthread_attr_setschedparam(&attr, &param);

        // Ensure the thread attributes are applied
        pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    }

    tinfo[i].thread_num = i ;// Thread numbering starts at 0
    tinfo[i].time_wait = time_wait;
    tinfo[i].sched_policy = policies[i];
    tinfo[i].sched_priority = priorities[i];


    // Create thread with the attributes
    int ret = pthread_create(&tinfo[i].thread_id, &attr, &thread_func, &tinfo[i]);

    set_thread_affinity(tinfo[i].thread_id, cpu_id);

    // Destroy the thread attribute object
    pthread_attr_destroy(&attr);
}
```

- Configuring CPU Affinity

  This task involves setting the CPU affinity for each thread, specifying which CPU core each thread should execute on. This helps control the thread's execution on a specific processor.

```
void set_thread_affinity(pthread_t thread, int cpu_id) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu_id, &cpuset);

    int ret = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
}
```

- Assigning Attributes to Each Thread

  Here, attributes like scheduling policy and priority are assigned to individual threads. These attributes define how threads are scheduled and their relative priorities.

```
pthread_attr_t attr;
struct sched_param param;

/* 2. Create <num_threads> worker threads with specific policies and priorities */
for (int i = num_threads - 1; i >= 0; i--)  {
    // Initialize thread attribute
    pthread_attr_init(&attr);

    //printf("policies[%d] = %d priority = %d \n", i, policies[i], priorities[i]);

    // Set the scheduling policy in the attribute structure
    pthread_attr_setschedpolicy(&attr, policies[i]);

    // Set the scheduling priority in the attribute structure only if it's not SCHED_NORMAL
    if (policies[i] != SCHED_OTHER) {
        param.sched_priority = priorities[i];
        pthread_attr_setschedparam(&attr, &param);

        // Ensure the thread attributes are applied
        pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    }

    tinfo[i].thread_num = i ;// Thread numbering starts at 0
    tinfo[i].time_wait = time_wait;
    tinfo[i].sched_policy = policies[i];
    tinfo[i].sched_priority = priorities[i];


    // Create thread with the attributes
    int ret = pthread_create(&tinfo[i].thread_id, &attr, &thread_func, &tinfo[i]);

    set_thread_affinity(tinfo[i].thread_id, cpu_id);

    // Destroy the thread attribute object
    pthread_attr_destroy(&attr);
}
```

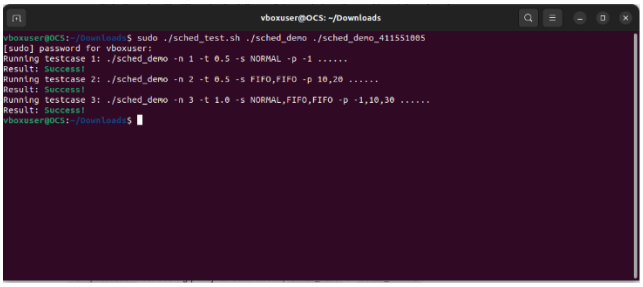
- Simultaneously Initiating All Threads
  This step ensures that all threads start their execution at roughly the same time. A synchronization barrier is used to make sure that no thread proceeds until all are ready to begin.

```
pthread_barrier_wait(&start_barrier);
```

- Waiting for All Threads to Complete Execution
  After starting the threads, the program waits for each thread to finish its work before proceeding further. This ensures that all threads complete their tasks before the program exits.

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

- Test Result



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

```
Thread 2 is running
Thread 2 is running
Thread 2 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 0 is running
Thread 0 is running
Thread 0 is running
```

There are three defined threads: Thread 1 and Thread 2 are configured as real-time tasks using the FIFO scheduling policy. Thread 2, having a higher priority than Thread 1, is executed before Thread 1. Meanwhile, Thread 0 remains suspended until all real-time tasks are completed.

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 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
```

There are four threads designated for execution. Among them, Threads 1 and 3 are configured as real-time tasks with the FIFO scheduling policy. Thread 3, being of higher priority than Thread 1, takes precedence in execution over Thread 1. On the other hand, Threads 0 and 2 are kept in a pending state until all real-time tasks have finished their execution. These two threads operate under the SCHED_OTHER policy, commonly referred to as the Completely Fair Scheduler (CFS). Consequently, Threads 0 and 2 alternate in executing for specified durations before yielding to other threads.

4. Describe how did you implement n-second-busy-waiting?
This busy_wait function essentially blocks the program's execution until the desired time has passed, and it does so by repeatedly checking the current time and comparing it to the start time until the desired waiting period has elapsed. It's a simple form of a busy-wait or sleep function that can be used for short delays in a program.