

Operating System - Assignment 2

Scheduling Policy Demonstration Program

I. Describe how you implemented the program in detail.

i. Parse program arguments

```
int num_threads = 0;
float time_wait = 0;
char *policies_str = NULL;
char *priorities_str = NULL;

int opt;
while((opt = getopt(argc, argv, "n:t:s:p:")) != -1){
    switch(opt){
        case 'n':
            num_threads = atoi(optarg);
            break;
        case 't':
            time_wait = atoi(optarg);
            break;
        case 's':
            policies_str = optarg;
            break;
        case 'p':
            priorities_str = optarg;
            break;
        default:
            //exit(EXIT_FAILURE);
            cout << endl;
    }
}

char *token = strtok(priorities_str, ",");
int priorities[num_threads];
for(int i = 0; i < num_threads; i++){
    priorities[i] = atoi(token);
    token = strtok(NULL, ",");
}

token = strtok(policies_str, ",");
int policies[num_threads];
for(int i = 0; i < num_threads; i++){
    if(token[0] == 'F'){
        policies[i] = 1;
    }
    else{
        policies[i] = 0;
    }
    token = strtok(NULL, ",");
}
```

- I use `getopt()` to parse incoming parameters. Among them, `n` and `t` are integers, and `s` and `p` are strings. Subsequently, I use `strtok()` to tokenize `s` and `p`, and then store them as array.
- For `s` (i.e. policies), I use 0 represent FIFO and 1 represent NORMAL.

ii. Create `<num_threads>` worker threads

```
pthread_t threads[num_threads];  
pthread_attr_t thread_attrs[num_threads];  
thread_info_t thread_infos[num_threads];  
pthread_barrier_init(&barrier, NULL, num_threads);
```

- Create the array to store the threads and its information.

iii. Set CPU affinity

```
cpu_set_t cpuset;  
CPU_ZERO(&cpuset);  
CPU_SET(i, &cpuset);  
pthread_setaffinity_np(threads[i], sizeof(cpu_set_t), &cpuset);
```

- Set cpu affinity.

iv. Set the attributes to each thread

```
for(int i = 0; i < num_threads; i++){  
    cpu_set_t cpuset;  
    CPU_ZERO(&cpuset);  
    CPU_SET(i, &cpuset);  
    pthread_setaffinity_np(threads[i], sizeof(cpu_set_t), &cpuset);  
  
    thread_infos[i].thread_num = i;  
    thread_infos[i].sched_policy = policies[i];  
    thread_infos[i].sched_priority = priorities[i];  
    thread_infos[i].time_wait = time_wait;  
  
    pthread_attr_init(&thread_attrs[i]);  
    if(thread_infos[i].sched_policy == 0){  
        pthread_attr_setschedpolicy(&thread_attrs[i], SCHED_OTHER);  
    }  
    else{  
        pthread_attr_setschedpolicy(&thread_attrs[i], SCHED_FIFO);  
    }  
    struct sched_param param;  
    param.sched_priority = thread_infos[i].sched_priority;  
    pthread_attr_setschedparam(&thread_attrs[i], &param);  
}
```

- Set the attributes of all the threads. Set their schedule policies and priorities.

v. Start all threads at once

```
for(int i = 0; i < num_threads; i++){  
    pthread_create(&threads[i], &thread_attrs[i], thread_func, (void *)&thread_infos[i]);  
}
```

```
void *thread_func(void *arg){  
    thread_info_t *thread_info = (thread_info_t *)arg;  
    int thread_id = thread_info -> thread_num;  
  
    pthread_barrier_wait(&barrier);  
    for(int i = 0; i < 3; i++){  
        printf("Thread %d is running\n", thread_id);  
    }  
  
    time_t time_start = time(NULL);  
    while(time(NULL) - time_start < thread_info -> time_wait){  
    }  
  
    pthread_exit(NULL);  
}
```

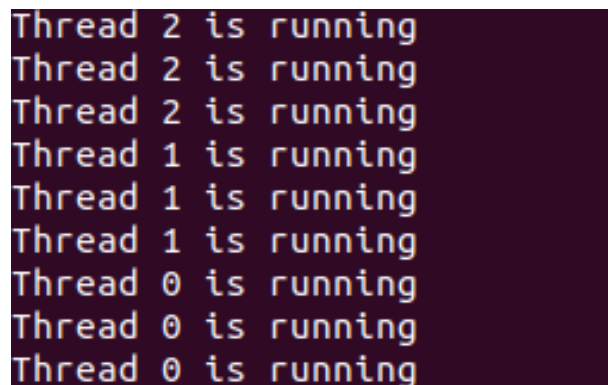
- Pass the configured parameters, including n and t as integers, and s and p as strings, to the pthread_create() function along with the task you want the threads to perform. Subsequently, create the threads, allowing them to execute the specified tasks. This involves setting up the necessary thread attributes, providing the function pointer to the task, and passing the required arguments. The threads will then run concurrently, each handling its designated portion of the workload based on the provided parameters.
- In thread_func(), thread will wait every threads by barrier.

vi. Wait for all threads to finish

```
for(int i = 0; i < num_threads; i++){  
    pthread_join(threads[i], NULL);  
}  
  
pthread_barrier_destroy(&barrier);
```

- By pthread_join(), threads will wait for others. After all threads have completed their tasks, the operating system will automatically reclaim and manage the associated resources.
- And we will destroy the barrier.

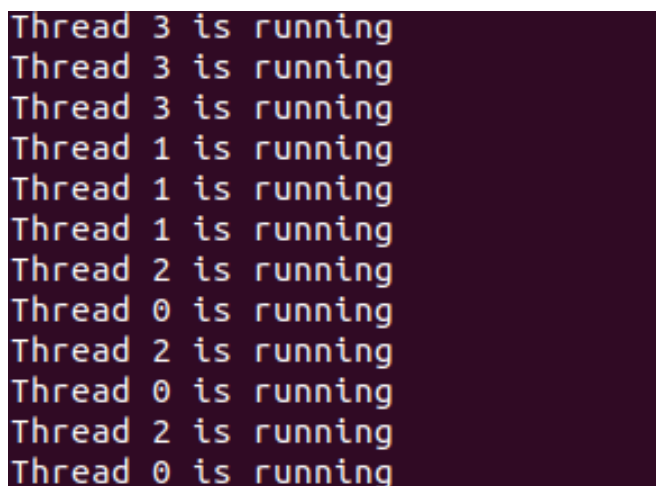
II. Describe the results of `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` and what causes that.



```
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
```

- Due to thread2 having a higher priority (priority 30) and being scheduled using the FIFO scheduling policy, it is selected to run before other threads. As a result, thread2 completes first, followed by thread1 (priority 10), and finally thread0 (priority -1).

III. 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
```

- Threads 3(30) and 1(10) have higher priority, so they will execute first. Subsequently, since thread 0 and thread 2 are using the Completely Fair Scheduler (CFS), they will be executed fairly.

IV. Describe how did you implement n-second-busy-waiting?

```
time_t time_start = time(NULL);  
while(time(NULL) - time_start < thread_info -> time_wait){  
}
```

- I employ a while loop to achieve this. I use a time function to make the thread repeatedly check whether the time requirement has been met. If the time has not yet arrived, the while loop must continue to execute, continuously evaluating the if `(time(NULL) - time_start < thread_info -> time_wait)` condition.