Assignment 2: Scheduling Policy Demonstration Program

Program source part

```
lunna@lunna:~/hw2$ sudo ./sched_test.sh ./sched_demo ./sched_demo_312555008
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!
```

The program implementation(20%)

Main function

Parse program arguments

Giving the example execute command line:

> *sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30*

There are many arguments need to parse,

So I use `getopt` to analyze the comment, and parse argument for each application

- `n <num_threads>`: number of threads to run simultaneously
- `t <time_wait>`: duration of “busy” period
- `s <policies>`: scheduling policy for each thread, `SCHED_FIFO` or `SCHED_NORMAL`.
- `p <priorities>`: real-time thread priority for real-time threads

```
/* 1. Parse program arguments */
    while ((ch = getopt(argc,argv,"n:t:s:p:"))!=-1)
    {
        switch(ch){
            case 'n':
                max_thread = atoi(optarg);
                break;
            case 't':
                buzy_period = strtod(optarg,NULL);
                break;
            case 's':
                strcpy(policies,optarg);
                break;
            case 'p':
                strcpy(priority,optarg);
                break;
            default:
                break;
        }
    }
```

Create <num_threads> worker threads

The struct of worker thread `thread_info_t` refer to which mentioned in the assignment.

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

In main(), declare the thread from `thread_info_t` called `th` , and malloc the space based on `max_thread` .

Then give `policy, priority, buzy_period` to each thread which are from the command parsed before.

- `Sched_NORMAL` : sched_policy=**0**
- `Sched_FIFO` : sched_policy=**1**

```
/* 2. Create <num_threads> worker threads */
                thread_info_t *th;
    th = malloc(max_thread*sizeof(thread_info_t));

    char *token;
    int t=0;
    token = strtok(policies, ",");
    while( token != NULL ) {
        if(strcmp("NORMAL",token)==0)
            th[t].sched_policy=0;
        else
            th[t].sched_policy=1;
        t++;
        token = strtok(NULL, ",");
    }
    t = 0;
    token = strtok(priority, ",");
    while( token != NULL ) {
        th[t].sched_priority = atoi(token);
        th[t].buzy_period = buzy_period;
        t++;
        token = strtok(NULL, ",");
    }
```

Set CPU affinity

Define all threads run in the same CPU which is `CPU_0` , and use `pthread_setaffinity_np()` to set CPU affinity to all threads.

```
/* 3. Set CPU affinity */
                int cpu_id = 0;
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu_id, &cpuset);
    sched_setaffinity(getpid(), sizeof(cpuset), &cpuset);
```

Set the attributes to each thread

If thread's scheduing policy is `SCHED_FIFO` , then set the attributes (`Policy, Priority`).

- `pthread_attr_setinheritsched` : Set inherit-scheduler attribute in thread attributes object
    - `PTHREAD_EXPLICIT_SCHED` : Set not inherit from main thread
- `pthread_attr_setschedparam` : Set scheduling parameters
- `pthread_attr_setschedpolicy` : Set scheduling policy

If thread's scheduing policy is `SCHED_NORMAL` , then it goes the fair scheduing policy

The default scheduling policy, so set `pthread_create()` `**attr**` be `NULL`

```
for (int i = 0; i < max_thread; i++) {
        /* 4. Set the attributes to each thread */
        th[i].thread_num=i;
        if(th[i].sched_priority != -1){
            pthread_attr_init(&attr);
            struct sched_param sp ;
            sp.sched_priority = th[i].sched_priority;
            pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
            pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
            pthread_attr_setschedparam(&attr, &sp);
            pthread_create(&th[i].thread_id, &attr, thread_func, (void *)(th+i));
        }
        else{
            pthread_create(&th[i].thread_id,NULL, thread_func, (void *)(th+i));
        }
    }
```

Synchronize threads

Use `pthread_barrier_wait` to let all threads start once; without this function, each thread will start when create, and it may make wrong result.

We need to `pthread_barrier_init` to initialize the barrier first with the **number of worker threads+1**.

Then put `pthread_barrier_wait()` after the `pthread_create()` and the initial of the `thread_func()`.

After thread finished, run `pthread_barrier_destroy()` to end the barrier.

```
**pthread_barrier_t barrier;**
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++) {
                                ...
            }
    }
    /* 3. Exit the function  */
    return NULL;
}
int main(){
                ...
                **pthread_barrier_init(&barrier, NULL, max_thread+1);**
    for (int i = 0; i < max_thread; i++) {
        ...
    }
    /* 5. Start all threads at once */
    **pthread_barrier_wait(&barrier);**

                for (int j = 0; j < max_thread; j++)
        pthread_join(th[j].thread_id, NULL);

                **pthread_barrier_destroy(&barrier);**
}
```

Wait for all threads to finish

Put `pthread_join()` waiting for all threads to finish.

```
for (int j = 0; j < max_thread; j++)
        pthread_join(th[j].thread_id, NULL);
```

Worker Thread Funciton

Put `pthread_barrier_wait` first for waiting until all threads are ready

Then make each thread run three times, print `Thread %d is running` with the `thread_num` every time. After print, wait for the buzy time then run the next round.

```
void *thread_func(void *arg)
{
    thread_info_t *my_data  = (thread_info_t*)arg;
    int thread_n = my_data->thread_num;
    double bzp = my_data->buzy_period;

    /* 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 running\n", thread_n);
        double sttime = my_clock();
            while (1) {
            if (my_clock() - sttime >= bzp)
                break;
            }
    }
    /* 3. Exit the function  */
    return NULL;
}
```

1. ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30

Describe the results and what causes that. (10%)

```
lunna@lunna:~/hw2$ sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
[sudo] password for lunna:
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
```

For `NORMAL`, its without priority number, goes default setting, which is **<1**

For `FIFO`, range from 1-99, **the lager number gets the higher priority**.

In this result we fund `Thread2` priority **30** is the larger than others, therefore it runs the first, and others need to wait for `Thread2` until it finished.

Same as `Thread2`, `Thread1` get priority **10**, which is larger than Thread 0, it runs when `Thread2` finished.

Last `Thread0` runs when other threads finished.

2. ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30

Describe the results and what causes that. (10%)

```
lunna@lunna:~/hw2$ sudo ./sched_demo -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
```

From above mentioned, `FIFO` will start with larger priority first, so `Thread3` (priority= 30) go first, `Thread1` started after.

`Thread0 & 2` are both `NORMAL`, it is up to the operating system scheduler to decide which one

gets scheduled to run first. Each thread give the same time to run according to CFS definition. After start, they took turns running, and the result be Thread2,0,2,0,…

3.Describe how did you implement n-second-busy-waiting? (10%)

```
static double my_clock(void) {
        struct timespec t;
        assert(clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) == 0);
        return 1e-9 * t.tv_nsec + t.tv_sec;
}

void *thread_func(void *arg)
{
    thread_info_t *my_data  = (thread_info_t*)arg;
    double bzp = my_data->buzy_period;

                ...

    /* 2. Do the task */
    for (int i = 0; i < 3; i++) {
                                ...
                    double sttime = my_clock();
        while (1) {
        if (my_clock() - sttime >= bzp)
            break;
        }
    }
    /* 3. Exit the function  */
    return NULL;
}
```

I wirte `my_clock()` to catch the time now with `clock_gettime()`.

```
assert(clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t) == 0);
return 1e-9 * t.tv_nsec + t.tv_sec;
//calculate to nano_second + second
```

It will calculate the passing time. When `current time - start time(sttime) ≥ setting time(bzp)`, it will go to next round.

```
double sttime = my_clock();
while (1) {
    if (my_clock() - sttime >= bzp)
        break;
}
```