YYYYYYY · Follow
Last edited by YYYYYYY on Nov 25, 2023
Contributed by

Assignment 2: Scheduling Policy Demonstration Program

This assignment aims to implement a program to apply different scheduling policies on created threads and observe their behaviors.

Linux Scheduling Policy

The scheduling polices can be divided into four categories:

- Fair scheduling policies
  - `SCHED_NORMAL` (CFS, `SCHED_OTHER` in POSIX), `SCHED_BATCH`
- Real-Time scheduling policies
  - `SCHED_FIFO`, `SCHED_RR`
- The other two are idle scheduling policy (`SCHED_IDLE`) and deadline scheduling policy (`SCHED_DEADLINE`)

The default scheduling policy is `SCHED_NORMAL` .

As you are only required to set either of `SCHED_NORMAL` or `SCHED_FIFO` policy to a thread in this assignment, and `SCHED_NORMAL` has been covered in the course, we will introduce the real-time scheduling policy `SCHED_FIFO` here.

`SCHED_FIFO`

What exactly does `SCHED_FIFO` do?

- `SCHED_FIFO` is a simple real-time scheduling algorithm **without time slicing**.
- When a `SCHED_FIFO` thread becomes runnable, it will **always immediately preempt** any currently running thread with fair scheduling policy.

So, when will a `SCHE_FIFO` thread be scheduled?

1. Blocked by an I/O request (becomes `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`)
2. Preempted by other `SCHED_FIFO` thread with higher priority.
   - When the thread is in **ready state**.
3. Calls `sched_yield(2)` or `sleep(3)` to give up the CPU resource.

You can run `ps -eo state,uid,pid,ppid,rtprio,time,comm` to list processes with their real-time priorities:

```
$ ps -eo state,uid,pid,ppid,rtprio,time,comm
S   UID     PID    PPID RTPRIO     TIME COMMAND
I     0       3       2      -  00:00:00 rcu_tasks_rude_kthread
I     0      14       2      -  00:00:00 rcu_tasks_trace_kthread
S     0      15       2      -  00:00:00 ksoftirqd/0
I     0      16       2      1 00:00:03 rcu_preempt
S     0      17       2      1 00:00:00 rcub/0
S     0      18       2     99 00:00:00 migration/0
S     0      19       2     50 00:00:00 idle_inject/0
```

The `RTPRIO` represents "real-time policy priority", which ranges from 1 to 99. The process with the smallest value 1 has the lowest priority. On the other hand, the processes with "-" in their `RTPRIO` column means they are not real-time processes (or threads).

As you can see, the processes `rcu_preempt` , `rcub/0` , `migration/0` and `idle_inject/0` are all real-time processes with their priority values.

Reference

- sched(7) - Linux manual page

Requirements

In this assignment, you are required to implement a program called `sched_demo` , which lets a user run multiple threads with different scheduling policies and show the working status of each thread.

```
$ 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
```

The meanings of command-line arguments are:

- `-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` .
  - The example `NORMAL,FIFO,NORMAL,FIFO` shown above means to apply `SCHED_NORMAL` policy to the 1st and 3rd thread and `SCHED_FIFO` policy to the 2nd and 4nd thread.
- `-p <priorities>` : real-time thread priority for real-time threads
  - The example `-1,10,-1,30` shown above means to set the value `10` to the 2nd thread and value `30` to the 4nd thread.
  - You should specify the value `-1` for threads with `SCHED_NORMAL` policy.

The program can be divided into two sections: the main thread section and worker thread section.

Main Thread

The main thread first needs to parse the program arguments, sets CPU affinity of all threads to **the same CPU**, and then creates `<num_threads>` worker threads specified by the option `-n` .

For each worker thread, attributes such as scheduling inheritance, scheduling policy and scheduling priority must be set. Next, the main thread will start all threads at once, and finally wait for all threads to complete.

```
int main() {
    /* 1. Parse program arguments */

    /* 2. Create <num_threads> worker threads */

    /* 3. Set CPU affinity */

    for (int i = 0; i < <num_threads>; i++) {
        /* 4. Set the attributes to each thread */
    }
    /* 5. Start all threads at once */

    /* 6. Wait for all threads to finish  */
}
```

Hint1 💡 : `getopt(3)` can be used to parse command-line options.

Hint2 💡 : Some useful functions:

- `sched_setaffinity(2)` / `pthread_setaffinity_np(3)` : Set CPU affinity
- `sched_setparam(2)` / `pthread_attr_setschedparam(3)` : Set scheduling parameters
- `sched_setscheduler(2)` / `pthread_attr_setschedpolicy(3)` : Set scheduling policy

Hint3 💡 : Start all threads one by one, like, in step 4, will have high probability producing incorrect outputs. You may want to see `pthread_barrier_wait(3p)` to synchronize threads.

Hint4 💡 :
Following is an example struct for collecting required thread information.

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

Worker Thread

Each newly-created worker thread must wait other threads before executing its task. After then, the thread runs the loop for three times.

In each loop, it shows a message indicating it's running and performs the busy work for `<time_wait>` seconds specified by the `-t` option. Finally, it exits the function.

```
void *thread_func(void *arg)
{
    /* 1. Wait until all threads are ready */

    /* 2. Do the task */
    for (int i = 0; i < 3; i++) {
        printf("Thread %d is running\n", <id-of-the-current-thread>);
        /* Busy for <time_wait> seconds */
    }
    /* 3. Exit the function  */
}
```

Warning ⚠️ : You can't use `sleep(3)` or `nanosleep(3)` functions to achieve busy working, which just cause the thread to become sleeping state and put it into the ready queue to be scheduled after the specified time. This explains why when you specify the sleep duration using `sleep` , but the actual sleep time is usually slightly more than the specified — one of the reasons is from the context-switching overhead.

Test

You can download the example program sched_demo(AMD) / sched_demo(ARM) and the test script sched_test.sh to test your program.

Note : If you cannot download them by clicking the link, please try the command `wget "the download link"`

We have re-uploaded a executable with the correct execution time, but the execution order between threads has not changed, so the correctness will not be affected. Students who have already submitted do not worry.

There are 3 testcases by default. If your program passes all testcases, it will show the message "Success!" for each testcase.

```
$ sudo ./sched_test.sh ./sched_demo ./sched_demo_<student_id>
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!
```

Note : If the command doesn't work, try these two commands
`chmod 777 sched_demo`
`chmod 777 sched_test.sh`

However, if your program fails any testcases, the test script will exit immediately and print the message "Failed..." with the `diff` results between two programs.

```
$ sudo ./sched_test.sh ./sched_demo ./sched_demo_<student_id>
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 .....
0a1,3
> Thread 1 is running
> Thread 1 is running
> Thread 1 is running
Result: Failed...
```

By the way, you can add your own testcases in the test script:

```
# You can add your own testcases here
testcases=("-n 1 -t 0.5 -s NORMAL -p -1"
           "-n 2 -t 0.5 -s FIFO,FIFO -p 10,20"
           "-n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30")
```

Hint 💡 : If you want to test your program alone, you may want to set `sysctl -w kernel.sched_rt_runtime_us=1000000` to ensure the real-time threads can fully utilize the CPU resource without being preempted by fair-time threads (**Warning: Only set it when testing as it may affect system processes!**)

Submission

Please submit a **zip** file to E3, which contains the program source and the report.

For the program source part (50%):

- The program must implemented using C or C++.
- Make sure your program passes all 3 testcases.
- Make sure your code can be compiled on **Ubuntu 22.04 AMD64 / ARM64**.

For the report part, you must answer the following questions (50%):

1. Describe how you implemented the program in detail. (20%)
2. Describe the results of `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` and what causes that. (10%)
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. (10%)
4. Describe how did you implement n-second-busy-waiting? (10%)

The name of the zip file should be `<student_id>.zip` , and the structure of the file should be as the following:

```
<student_id>.zip
    |- <student_id>/
        |- report_<student_id>.pdf
        |- sched_demo_<student_id>.c (or sched_demo_<student_id>.cpp)
```

The deadline is on **11/29 23:59**.