Operating System Homework 2

Team53: b05902121 黃冠博 b05902019 蔡青邑

May 17, 2018

part 1

I got couple error messages like this:

```
unknown type name ‘ pthread_barrier_t ’
implicit declaration of function ‘ sched_setaffinity ’ [-Wimplicit-function-declaration]
implicit declaration of function ‘ CPU_ZERO ’ [-Wimplicit-function-declaration]
implicit declaration of function ‘ CPU_SET ’ [-Wimplicit-function-declaration]
```

I went to stackoverflow and found out that a feature test macro must be defined before including any header files. In other words, I have to put **#define __GNU__SOURCE** in the beginning of my code.

I wrote a function for the sole argument **arg**.

```c
    void* arg_par(int thr_num, int sched_num){
        int tmp[2];
        tmp[0] = thr_num, tmp[1] = sched_num;
        return (void*) tmp;
    }
```

I got a warning like this:

```
warning: function returns address of local variable [enabled by default]
```

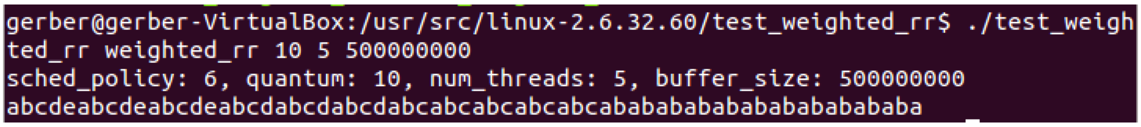
Hence, I adjust the code lines and moved them into the main function.

```c
    int *arg = (int *)malloc(2*sizeof(int));
    *arg = i + 1;
    *(arg+1) = default_sched;
    int rt = pthread_create(&thread_id[i],NULL,thread_func,(void*) arg);
```

Returning local addresses in functions is dangerous.

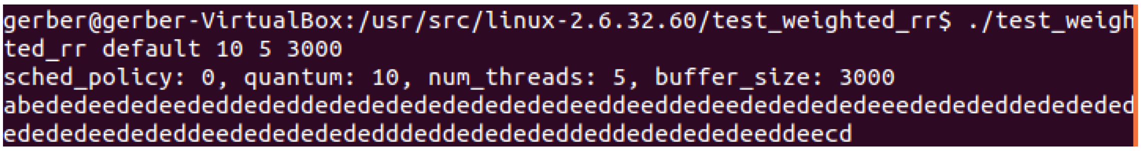
part 2

result



```
gerber@gerber-VirtualBox:/usr/src/linux-2.6.32.60/test_weighted_rr$ ./test_weigh
ted_rr weighted_rr 10 5 500000000
sched_policy: 6, quantum: 10, num_threads: 5, buffer_size: 500000000
abcdeabcdeabcdeabcdabcdabcdabcabcabcabcabcababababababababababa
```

We can see that e terminates earlier than d, d terminates earlier than c, and so on.
This is because e has a larger quantum than d. Since every thread has to write `total_num_chars / num_threads` of chars, threads with larger quantum need less rounds to finish.



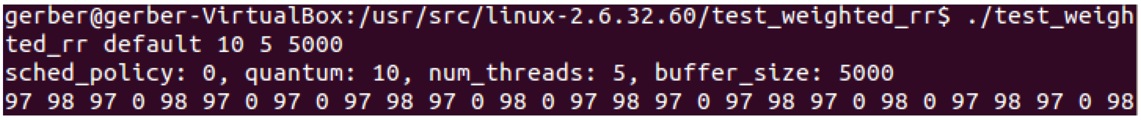
```
gerber@gerber-VirtualBox:/usr/src/linux-2.6.32.60/test_weighted_rr$ ./test_weigh
ted_rr default 10 5 3000
sched_policy: 0, quantum: 10, num_threads: 5, buffer_size: 3000
abededeededeededdededdedededededededededeeddeeddedeededededededeeeedededeeddededededed
edededeedededddeededededededeedddeddedededeeddeddedededededeeddeecd
```

In `test_weighted_rr.c`, there is a piece of code:

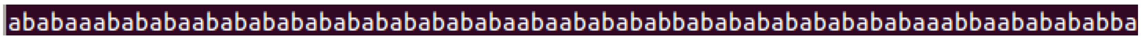
```
for (i = 0; i < total_num_chars; i++) {
    if (cur != val_buf[i]) {
        cur = val_buf[i];
        printf("%c", cur);
    }
}
```

I thought this code should ensure that no same characters are printed continuously, however, the results aren't as expected. Hence, I added my own code lines into `test_weighted_rr.c` as shown below and got an interesting result. (There are too many lines, so we just show the critical ones.)

```
for (i = 0; i < total_num_chars; i++) {
    if (cur != val_buf[i]) {
        cur = val_buf[i];
        printf("%d ", cur);
    }
}
```



```
gerber@gerber-VirtualBox:/usr/src/linux-2.6.32.60/test_weighted_rr$ ./test_weigh
ted_rr default 10 5 5000
sched_policy: 0, quantum: 10, num_threads: 5, buffer_size: 5000
97 98 97 0 98 97 0 97 0 97 98 97 0 98 0 97 98 97 0 97 98 97 0 98 0 97 98 97 0 98
```



```
ababaaabababaababababababababababaabaabababbababababababababaaabbaababababba
```

We can see that there are three 'a's continuously, however, it seems like the sequence isn't `97 97 97` but `97 0 97 0 97`. This means that there are characters `'\0'` between.

There is a piece of code line that I struggled in for a long time. As shown below:

```
for (i = 0; i < num_threads; i++){
    targs = malloc(sizeof(*targs));
    targs->tid    = i;
    targs->prio   = i;
    targs->mychar = (char) (i+START_CHAR);
    targs->nchars = (total_num_chars / num_threads);

    if(quantum <= i) printf("Time quantum too small\n");
    else syscall (SYS_weighted_rr_setquantum, quantum);

    pthread_create(&threads[i], &attr, run, (void *)targs);
    if (sched_policy == SCHED_WEIGHTED_RR) quantum*=2;
}
```

I was wondering that whether function `syscall (SYS_weighted_rr_setquantum, quantum)` will set the quantum for every thread the same, since it changes the quantum of the system every time before creating a new thread. However, the address of the variable specifying the quantum for each thread is different, so there are no overriding problems in this situation.

Implementing

`enqueue_task_weighted_rr()`:

use `list_add_tail` and update the value of `rq->weighted_rr.nr_running`

`dequeue_task_weighted_rr()`:

first `update_curr_weighted_rr(rq)`

use `list_del` and update the value of `rq->weighted_rr.nr_running`

`yield_task_weighted_rr()`:

call `requeue_task_weighted_rr`, it uses `list_move_tail` to put the current task `rq->curr` to the end of the running list

`pick_next_task_weighted_rr()`:

If list is empty, return `NULL`.

use `list_first_entry()` to get the first task in the list. It returns the next task.

set `next->se.exec_start = rq->clock`

`u64` `exec_start` is in `struct sched_entity`

`task_tick_weighted_rr()`:

`task_tick_weighted_rr` is invoked on each scheduler timer tick.
first update the task's runtime statistics
If the value of `task_time_slice` of task p is 0, reset `task_time_slice` of task p,
then use `set_tsk_need_resched(p)`.
Finally, use `requeue_task_weighted_rr` to put task p to the end
of the running list without the overhead of dequeue followed by enqueue.