

EECS 678 Project II

Process Scheduling in Linux

Project Report
04/23/15

Group Member:
Wei Fei
Jiahui Wang

Introduction

In this project, we need to implement the **round robin scheduler** by using the thread runner program and **examine round robin scheduling in Linux**. Round Robin Scheduling method is a kind simple way to avoid starvation by using FCFS with preemption when multiple threads are running. The method schedules the first job in the queue for one time-slice preempt job and add it to the end of the queue, and then schedule the next job and continue again and again.

Before making change in the source codes, we should set up the custom kernel image on the KVM. And then we are supposed to edit the given source code to demonstrate the round robin scheduler. After that, we need to use some system calls to choose the schedulers. Finally, we should try to change the parameters and use different quantum number, thread number and value of size to measure the function of round robin scheduler with different result. For example, we should measure the program work well while the quantum is 0, which is the first coming first serving schedulers. Or change the quantum to 5, which will demonstrate the normal round robin scheduler. While we modify the source codes, we should make some changes in five specific files, which are “kernel/sched.c, kernel/sched_other_rr.c, yscall_table_32.S, unistd_32.h, and syscalls.h”.

Implementation

➤ kernel/sched.c

We need to add one more case like “`SCHED_OTHER_RR`” (line 6508 ~ 6511) firstly, and then add new function named “`SYSCALL_DEFINE1()`”, which allows us to pass the quantum to scheduler in the system call. Also print out the message to the system log when `SCHED_OTHER_RR` is selected.

Sample codes:

case `SCHED_OTHER_RR`:

```
p->sched_class = &other_rr_sched_class;
printk("SCHED_OTHER_RR is selected\n");
break;
```

```

SYSCALL_DEFINE1(sched_other_rr_setquantum, unsigned int, quantum)
{
    other_rr_time_slice = quantum;
    printk("scheduler receives the quantum\n");
    return 0;
}

```

➤ kernel/sched_other_rr.c

- **Inserting/Removing (enqueue and dequeue):** In the adding/removing function, we should update the current tasks runtime statistics, and add the tasks to tail of the queue, and we also need to maintain the count of running process when we insert or remove. For the function of removing, it is similar to the inserting function, just do some opposed methods.

Sample codes:

```

static void enqueue_task_other_rr(struct rq *rq, struct task_struct *p, int wakeup,
bool b)
{
    update_curr_other_rr(rq);
    list_add_tail(&p->other_rr_run_list, &rq->other_rr.queue);
    rq->other_rr.nr_running++;
}

static void dequeue_task_other_rr(struct rq *rq, struct task_struct *p, int sleep)
{
    update_curr_other_rr(rq);
    list_del(&p->other_rr_run_list);
    rq->other_rr.nr_running--;
}

```

- **yield_task_other_rr:** we should invoke the function “requeue_task_other_rr();” to put the task to the end of the run list without the overhead of dequeue followed by enqueue.

Sample codes:

```

static void yield_task_other_rr(struct rq *rq)

```

```

{
    requeue_task_other_rr(rq, rq->curr);
}

```

- **pick_next_task_other_rr:** (Selecting next task supposed to run): we need to pick the first element of the queue, and maintain the correct runtime statistics. In other words, we should update the execution start time of current task, and return a pointer to the task or return null if there are nothing in queue.

Sample codes:

```

if(list_empty(queue))
{
    next = NULL;
}
else
{
    next = list_first_entry(queue, struct task_struct, other_rr_run_list);
    next->se.exec_start = rq->clock;
}

```

- **task_tick_other_rr (timer tick):** when the other_rr time slice is 0, we will run the FCFS scheduler; when the other_rr time slice larger than 0, and task time slice is not equal to 0, reduce the task time slice; otherwise, we should change the task time slice to current other_rr time slice, moving the task to the end of queue, and invoke “set_tsk_need_resched()”.

Sample codes:

```

if(other_rr_time_slice == 0)
{
    return;
}
else
{
    if(p->task_time_slice != 0)
    {

```

```

        p->task_time_slice--;
    }
    else
    {
        p->task_time_slice = other_rr_time_slice;
        requeue_task_other_rr(rq, p);
        set_tsk_need_resched(p);
    }
}

```

➤ **“syscall_table_32.S”, “unistd_32.h”, “syscalls.h”**

Adding system call to set the quantum in the files:

- **“syscall_table_32.S”**: we add the system call named “long sys_sched_other_rr_getquantum” and “long sys_sched_other_rr_setquantum” in the end of file.
- **“unistd_32.h”**: we add “#define __NR_sched_other_rr_setquantum 338”(line 346), and increase the later NR syscalls by 1.
- **“syscalls.h”**: we add two declaration for the system calls: “sys_sched_other_rr_

➤ **Finally**, we need to test the system calls to make sure the added system calls work correctly.

After that using the provided thread runner program to test schedulers. And we will use different quantum and different threads combinations in the schedulers testing.

Discussion and Error Analysis

➤ **Pass by Value and Pass by Reference**

When we was implemeting the enqueue task function and dequeue task function in “sched_other_rr.c” file, we forgot to add “&” before passing parameter. It leads to some errors. After we see the error line number, we find the similarities of these errors. So we guess it is probably caused by the difference between “passing my value” and “passing by reference”. For example, for the function “enqueue_task_other_rr”, we need to invoke the function “list_add_tail”, which needs to insert a new entry before the specified head. For this function, we have to pass by reference for the parameters “new” and “head”, since the original function's declaration is “list_add_tail (struct list_head * new, struct list_head * head)”, which the parameters are initialized to the type of pointer. If we only pass by value without the symbol “&”, the memory address of the parameter will not be passed to the original function. After we fix these passing types, the errors were removed.

➤ **Quantum type problem**

When we was editing the file “sched.c”, we made a mistake in the function “SYSCALL_DEFINE1”. For the second parameter of this function, we need to pass the item type. Previously, we put “int” on the slot of this parameter, but it is not correct. Because the quantum number only can be a number greater than 0, which means we need to use “unsigned int” to initialize the variable “quantum”. After we fix it, the function is correct.

➤ **Running command problem**

After we completing to compile all codes and to build the kernel, we need to run the whole program. Originally, we typed the command “./thread_runner -quantum (quantum number) -s other_rr (size)”, it shows me some hints to remind me we forgot to type the parameter “num_threads”. So we typed “./thread_runner -quantum (quantum number) -s other_rr (thread number) (size)” to run the program again. However, it still shows me segmentation fault. After my friend and we asking our instructor, we know the problem is because we miss one dash before “-quantum”. The correct command should be “./thread_runner --quantum (quantum number) -s other_rr (thread number) (size)”. After we fixing this problem, we got the correct output.

Difficulties

We think the coding part of this project is much easier than previous one. However, the **building process and running process** are the biggest challenge for us since it is the first time we touch the kvm and thread runner. We need to exposeIn addition, **debugging procedure** for this project is too complex, we try to use debugger that we learned in lab section, but we do not think we can handle it. So the way to debug of us is that to edit in gedit editor when we have errors and use “scp” command to copy our updated codes into kvm virtual machine, and then build the program again. We also put a lot of “printf()” function to check where the error is. I think the debug procedure are more complicated and more inconvenient than the normal program that we have learned before.

Conclusion

Round Robin Scheduling method is a kind simple way to avoid starvation by using **FCFS with preemption** when multiple threads are running. The method schedules the first job in the queue for one time-slice preempt job and add it to the end of the queue, and then schedule the next job and continue again and again. The advantage of round robin method is better interactivity, and disadvantage is that the performance is dependent on the quantum size.

From the different outputs with different parameters, we find that the **quantum number is the most significant variable** to decide the result of round robin scheduler. If the quantum number is too big or equal to 0, it is FCFS like behavior. On the contrary, if the quantum number is too small, it may lead to large context switch overhead. So choose an appropriate quantum number is important for replying advantages of round robin scheduler method.

Sample Outputs

Remark: we test some results with different variables by using the method “controlling variables”.

- ### 1. Quantum = 0, num_thread = 6, size = 5000k (FCFS)

[If the quantum number equals to 0, which means it equivalent to the scheduling method of “first come first serve”]

[illegible]

2. Quantum = 5, num_thread = 6, size = 5000k (RR)

[We choose a middle quantum number, which may become an appropriate quantum number]

```
wfei26@kvm: ~/thread_runner
File Edit View Search Terminal Tabs Help

wfei26@cycle1:~ wfei26@kvm: ~/thread_runner

wfei26@kvm:~/thread_runner$ ./thread_runner --quantum 5 -s other_rr 6 5000k
other_rr scheduler selected, quantum=5
thread: 0 writing 853333 a's
thread: 1 writing 853333 b's
thread: 2 writing 853333 c's
thread: 3 writing 853333 d's
thread: 4 writing 853333 e's
thread: 5 writing 853333 f's

completed 6 threads -- processing shared memory segment

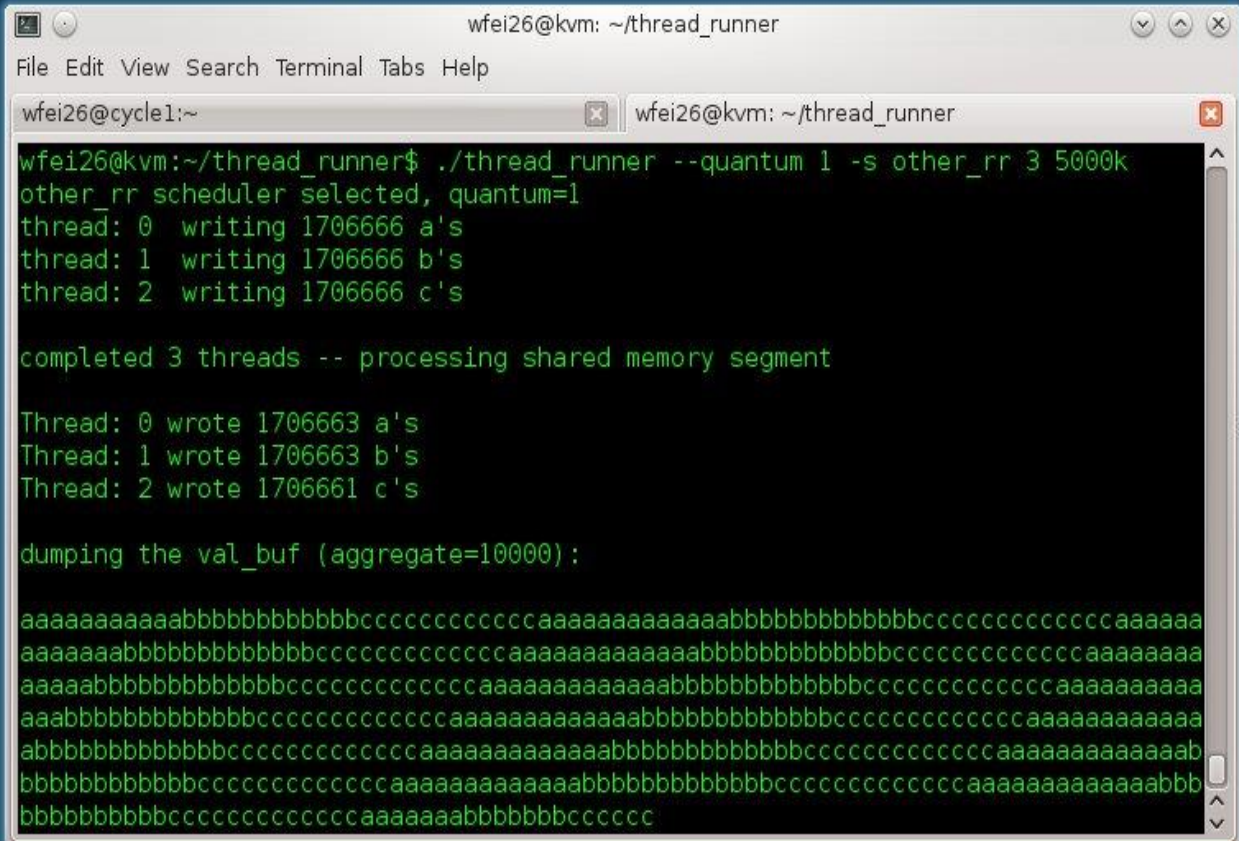
Thread: 0 wrote 853332 a's
Thread: 1 wrote 853333 b's
Thread: 2 wrote 853332 c's
Thread: 3 wrote 853332 d's
Thread: 4 wrote 853332 e's
Thread: 5 wrote 853332 f's

dumping the val_buf (aggregate=10000):

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccc
ccccccccccccccccccccccccccccccccccccddddddeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeefffffffaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccccccccccc
ccccccccccccccccccccccccccccccccccccddddddeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeefffffffaaaaaaaaaaaaaabbbbbbbbbb
ccccccccccccccccccccccccccccccccccccffffff
```

3. Quantum = 1, num_thread = 3, size = 5000k (RR)

[Choose 1 as quantum number, the round robin method may lose advantages because of inappropriate quantum number. Shorter response time, but more context switch]



```
wfei26@kvm: ~/thread_runner
File Edit View Search Terminal Tabs Help

wfei26@cycle1:~
wfei26@kvm: ~/thread_runner

wfei26@kvm:~/thread_runner$ ./thread_runner --quantum 1 -s other_rr 3 5000k
other_rr scheduler selected, quantum=1
thread: 0  writing 1706666 a's
thread: 1  writing 1706666 b's
thread: 2  writing 1706666 c's

completed 3 threads -- processing shared memory segment

Thread: 0 wrote 1706663 a's
Thread: 1 wrote 1706663 b's
Thread: 2 wrote 1706661 c's

dumping the val_buf (aggregate=10000):

aaaaaaaaaabbabbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
aaaaaabbabbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
aaaaabbabbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
aaabbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
abbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
bbbbbbbcccccccccaaaaaaaaaaabbabbbbbbcccccccccaaaaaa
bbbbbbbcccccccccaaaaaabbabbbbbbcccccc
```

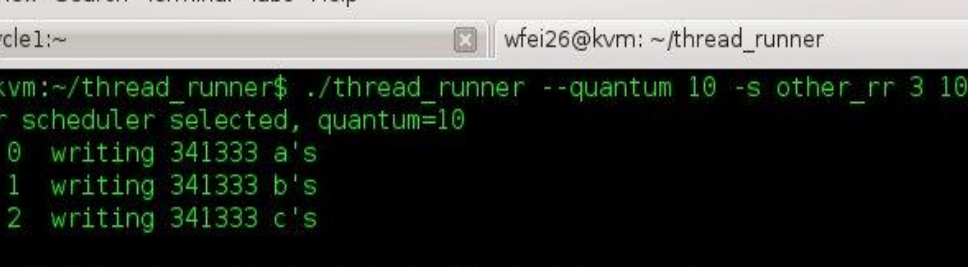
4. Quantum = 10, num_thread = 3, size = 5000k (RR)

[If the quantum number is large enough, every thread will almost finish its task in their first time period]

[illegible]

5. Quantum = 10, num_thread = 3, size = 1000k (RR)

[Keep the quantum number and thread number same as test 4, change the size to be smaller]



```
wfei26@kvm: ~/thread_runner
File Edit View Search Terminal Tabs Help

wfei26@cycle1:~
wfei26@kvm: ~/thread_runner

wfei26@kvm:~/thread_runner$ ./thread_runner --quantum 10 -s other_rr 3 1000k
other_rr scheduler selected, quantum=10
thread: 0  writing 341333 a's
thread: 1  writing 341333 b's
thread: 2  writing 341333 c's

completed 3 threads -- processing shared memory segment

Thread: 0 wrote 341333 a's
Thread: 1 wrote 341333 b's
Thread: 2 wrote 341333 c's

dumping the val_buf (aggregate=10000):

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccccc
cccccccccccccccccccccccccccccccccccc
```