Real Time System
Project#2

Team7 - R04922058 Yi Lin Cheng , R04922034 Hsuan-Heng, Wu

Part I: Weighted Round Robin Scheduling

Enqueue task

When a new task is created, it needs to be put into `weighted_rr_rq`, which is done by calling `list_add_tail(&H,&E)`. This function inserts a new entry E before the specify head H. Thus, given `weighted_rr_rq`'s head as H, E will be inserted at the end of the queue. After inserting E, increase counter `weighted_rr_rq.nr_running` by one.

Dequeue task

Deleting a specify task T is done by calling `list_del(&T.list_head)`. This function removes T from the list by modifying `T.list_head`'s pointers and the pointers pointing to `T.list_head`. After removing T, decrease counter `weighted_rr_rq.nr_running` by one.

Pick next task

This function is called when OS needs to select the next task to run. Since we want to implement a weighted round robin scheduler, this function should return the first element in `weighted_rr_rq`. We use `list_first_entry()` to retrieve the first element in run queue. Note that we should check if the queue is empty before trying to retrieve an entry. If `nr_running` equals 0, simply return `NULL`.

Task tick

Each task has its own time quantum. Every time when `test_tick_weighted_rr()` is called, the quantum of current running task P should be decreased. When a task exhausts its time quantum, the quantum will be replenished but it should yield the CPU and be rescheduled. There are two steps in this function:

1. Decrease P's time quantum: `p->task_time_slice--`.
2. If P's quantum is exhausted:
   Replenish P's time quantum: `p->task_time_clice=p->weighted_time_slice`.
   Move P to the end of the queue: `requeue_task_weighted_rr(rq, P)`.
   Specify P needs to be rescheduled: `set_tsk_need_resched(P)`.

Yield task

When a task yields the CPU, it should be moved to the end of run queue. When this function is called, it first retrieves the current running task T by `rq->curr`, then call `requeue_task_weighted_rr(&T)` to move T.

PART II: Shortest Job First Scheduling

The implemenation of sched_sjf.c is mostly based on the implemenation of sched_weighted_rr.c. Note that the initial `sjf_time_slice` is assumed to be the execution time of job.

Enqueue task

Same as part1.

Dequeue task

Same as part1.

Pick next task

In shortest job first scheduling implementation, the job with mimimum sjf_time_slice should be selected, thus a list_for_each call is used to traverse the `sjf->queue` to extract the task with minimum job execution time. We use `list_first_entry()` to retrieve the first element in run queue and get its corresponding time_slice as initial minimum execution time to be later compared throughout iterations. Note that we should check if the queue is empty before trying to retrieve an entry. If `nr_running` equals 0, simply return `NULL`.

Task tick

The implementation of this function is almost identical to that of part 1. All we need to do is to comment out the section of code where a task is rescheduled if it runs out of time slot because SJF is a no preemption protocol.

Yield task

Same as part 1.

Define New Schedule Policy

In include\linux\shed.h , define SCHED_SJF = 7

Define New System Call

In arch \x86 \include \asm \unistd_32.h:

1. define __NR_sched_sjf_getquantum 339
2. define __NR_sched_sjf_setquantum 340

3. define __NR_syscalls 343

In arch \x86 \kernel\syscall_table_32.S

1. append .long sys_sched_sjf_getquantum
2. append .long sys_sched_sjf_setquantum

In include \linux \syscalls.h

1. append asmlinkage long sys_sched_sjf_getquantum(void);
2. append asmlinkage long sys_sched_sjf_setquantum(unsigned int quantum);

Update Sched.c

include sched_sjf.c In kernel\sched_.c

1. add sjf_rq
2. define and implement sched_sjf_getquantum
3. define and implement sched_sjf_getquantum
4. define sjf_time_slice

Concatenate list of sched_classes

In kernel\sched_weighted_rr.c

1. update weighted_rr_sched_class.next = sjf_sched_class

In kernel\sched_sjf.c

1. update sjf_sched_class.next = idle_sched_class

Part III: Rate Monotonic Scheduling

The implementation of Part 3 is based on recompiliing the kernel with different version of sched_sjf.c using SCHED_SJF due to some unknown error that prevents SCHED_RMS to work. Thus , for sched_rms.c to work, it must be renamed to sched_sjf.c and replace the original one such that sched_sjf.c will perform different policies.

Enqueue task

Same as part 1.

Dequeue task

Same as part 1.

Pick next task

Same as part 2 , since we assign task we lower period higher priority.

Task tick

Similar to part 1, but since Rate Monotonic allows preemption, we need to check whether there are tasks with higher priority in the task_queue. This is done by a list_for_each iterative check and an additional or check for rescheduling.

Yield task

Same as part 1.

Part IV: Test Cases

For Part II

In the for loop that creates pthread, add random time quantum to the base time quantum , and set the last thread created to the highest possible priority (lowest execTime). The design of last thread with lowest execTime is used to test whether preemption is successfully disabled.

For Part III

Similar to the case for part II, except that a usleep call is performed before the creation of last thread, this design is used to test whether preemption can happen at every tick.

Result

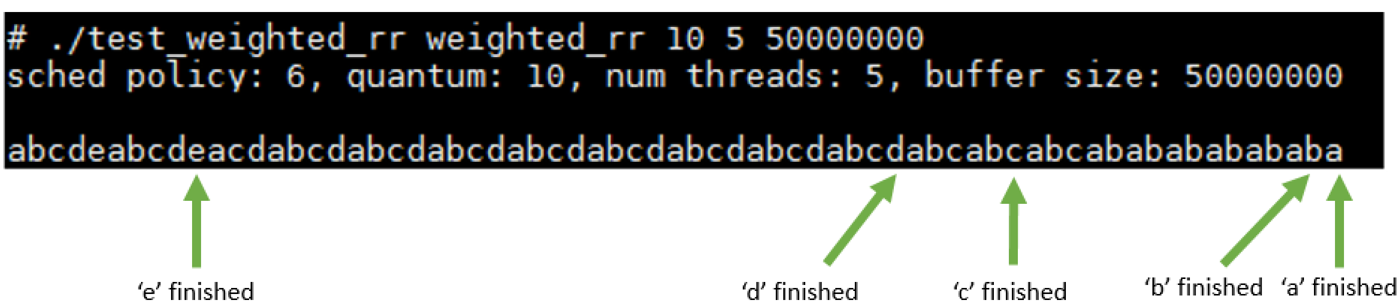

Figure 1: Result of Part I

```
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_sjf sjf 10 5 5000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 5000
a job exeTime=20
b job exeTime=23
c job exeTime=19
d job exeTime=18
e job exeTime=10
edcab
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_sjf sjf 10 5 50000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 50000
a job exeTime=10
b job exeTime=18
c job exeTime=25
d job exeTime=11
e job exeTime=10
aedbc
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_sjf sjf 10 5 500000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 500000
a job exeTime=20
b job exeTime=16
c job exeTime=11
d job exeTime=27
e job exeTime=10
ecbad
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_sjf sjf 10 5 5000000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 5000000
a job exeTime=14
b job exeTime=13
c job exeTime=18
d job exeTime=29
e job exeTime=10
ebacd
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_sjf sjf 10 5 50000000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 50000000
a job exeTime=12
b job exeTime=14
c job exeTime=27
d job exeTime=27
e job exeTime=10
eabcd
```

Figure 2: Result of Part II

```
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_rms rms 10 5 5000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 5000
a job exeTime=12
b job exeTime=14
c job exeTime=16
d job exeTime=12
e job exeTime=10
aedbc
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_rms rms 10 5 50000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 50000
a job exeTime=18
b job exeTime=27
c job exeTime=16
d job exeTime=20
e job exeTime=10
caedb
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_rms rms 10 5 500000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 500000
a job exeTime=26
b job exeTime=29
c job exeTime=10
d job exeTime=16
e job exeTime=10
cedab
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_rms rms 10 5 5000000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 5000000
a job exeTime=22
b job exeTime=28
c job exeTime=13
d job exeTime=16
e job exeTime=10
cecdab
rts@rts-VirtualBox:~/linux-2.6.32.60/test_weighted_rr$ ./test_rms rms 10 5 50000000
sched_policy: 7, quantum: 10, num_threads: 5, buffer_size: 50000000
a job exeTime=27
b job exeTime=10
c job exeTime=14
d job exeTime=19
e job exeTime=10
becda
```

Figure 3: Result of Part III