Project 2: Android Scheduler

Implementation of Weighted-Round-Robin

Dou Yiming (519021910366, douyiming@sjtu.edu.cn)

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

1 Introduction

CPU scheduling, which is the basis of modern operating systems, can make the computer execute much faster by switching CPU among different processes. In this project, we are required to implement a specific CPU scheduler **WRR** for Linux based Android Operaing System.

WRR is the abbreviation of Weighted-Round-Robin, which is a case distinct Round-Robin schedule policy. **WRR** assigns longer time slices for foreground tasks while assigns shorter time slices for background tasks.

2 Preliminary Analysis

To have a more profound understanding of the linux scheduler, especially RR scheduler, i firstly refer to the website provided by JINKYU KOO, and i further read lots of source code in core.c, rt.c, and sched.c.

After a long time of understanding and investigation, i get familiar to the schedulers and how they work. Moreover, my work is to implement a new scheduler that is very similar to rt and make it work like rt does.

3 Implementation of wrr.c

File wrr.c contains the major part of the whole project, which is wrr-sched-class.

The main procedure of implementing this class and the related functions is imitating and revising the implementation in rt.c.

- Firstly, when imitating, all of the codes related to SMP, preemption should be omitted, since these functionalities are not required.
- Secondly, some operations on struct should be substitued with built-in list opeations. For example, bitmap and prio-array should be modified.
- Thirdly, in order to achieve the ability of distinguishing foreground processes from background ones and further assign different time slices to them, cgroup-path is used to get the state. We add the checking of process state in task-tick-wrr and get-rr-interval-wrr.

4 Implementation of Other Files

4.1 Revision on /include/linux/sched.h

1. The definition of SCHED-WRR is shown in Figure 1:
2. The definition of sched-wrr-entity is shown in Figure 2:
3. The definition of time slices is shown in Figure 3:
4. Adding wrr-rq is shown in Figure 4:
5. The declaration of wrr-rq struct is shown in Figure 5:

4.2 Revision on /kernel/sched/sched.h

1. The declaration of wrr-rq struct is shown in Figure 1:
2. The definition of struct wrr-rq is shown in Figure 2:
3. Adding of wrr-rq variable and list-head variable to struct rq is shown in Figure 3:

```
#define SCHED_NORMAL        0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented y
#define SCHED_IDLE      5
#define SCHED_WRR       6
/* Can be ORed in to make sure the process i
#define SCHED_RESET_ON_FORK     0x40000000
```

Fig. 1. Definition of SCHED-WRR

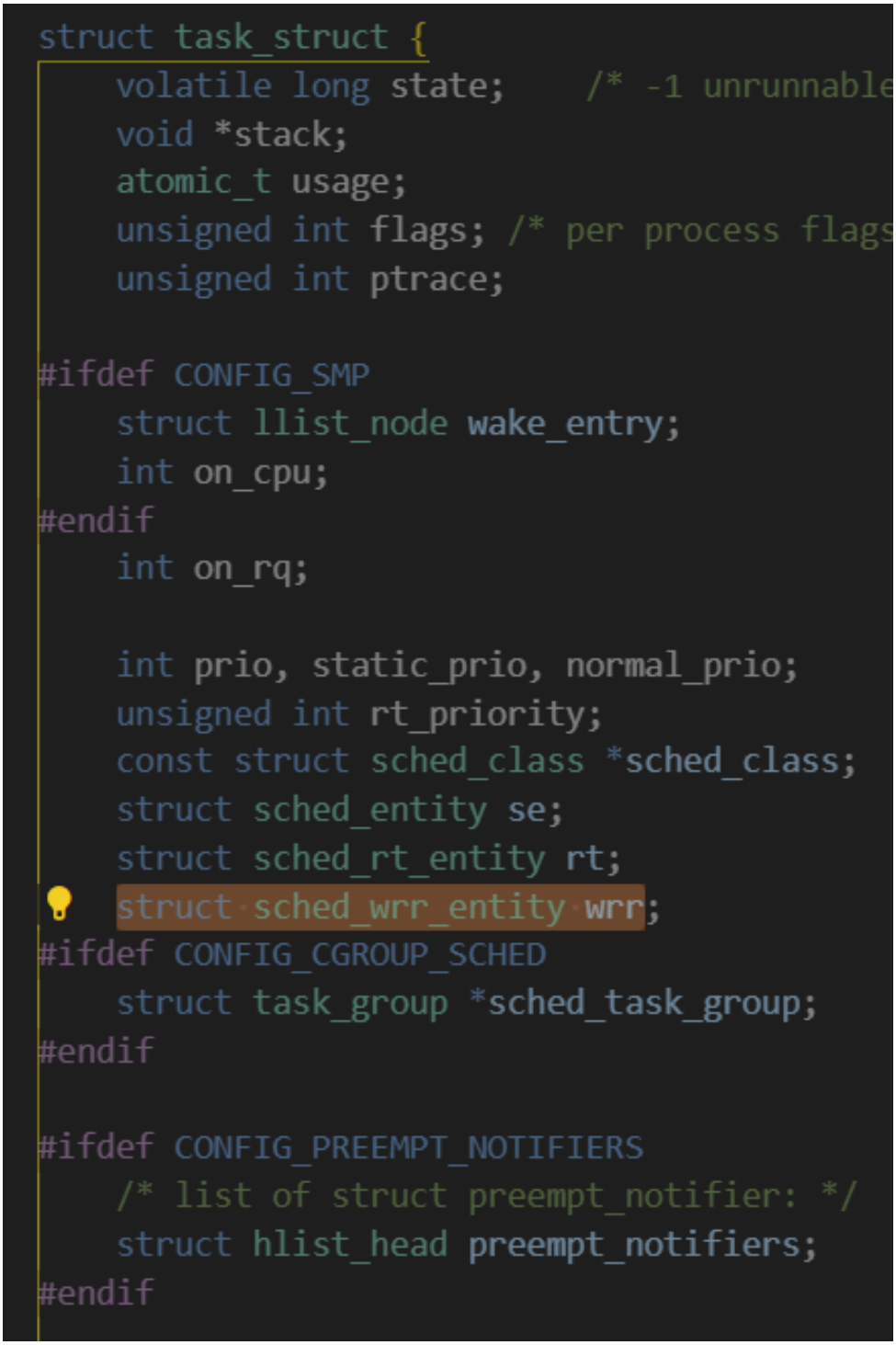
```
struct sched_wrr_entity
{
    struct list_head run_list;
    unsigned long timeout;
    unsigned int time_slice;
    int nr_cpus_allowed;

    struct sched_wrr_entity *back;
#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_wrr_entity *parent;
    /* rq on which this entity is (to be) queued: */
    struct wrr_rq       *wrr_rq;
    /* rq "owned" by this entity/group: */
    struct wrr_rq       *my_q;
#endif
};
```

Fig. 2. Definition of sched-wrr-entity

```
#define WRR_TIMESLICE_FORE      (100 * HZ / 1000)
#define WRR_TIMESLICE_BACK      (10 * HZ / 1000)
```

Fig. 3. Definition of time slices



```
struct task_struct {
    volatile long state;    /* -1 unrunnable
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    struct sched_wrr_entity wrr;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group *sched_task_group;
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
#endif
```

Fig. 4. Adding wrr-rq in task-struct

```
struct cfs_rq;
struct wrr_rq; // declaring struct
struct task_group;
```

Fig. 5. Declaration of struct wrr-rq

```
struct cfs_rq;
struct rt_rq;
struct wrr_rq; // newly-defined
```

Fig. 6. Definition of SCHED-WRR

```
struct wrr_rq
{
    struct list_head wrr_queue;
    unsigned long wrr_nr_running;
#if defined CONFIG_SMP
    struct {
        int curr; /* highest queued wrr task prio */
#ifdef CONFIG_SMP
        int next; /* next highest */
#endif
    } highest_prio;
#endif
#ifdef CONFIG_SMP
    unsigned long wrr_nr_migratory;
    unsigned long wrr_nr_total;
    int overloaded;
    struct plist_head pushable_tasks;
#endif
    int wrr_throttled;
    u64 wrr_time;
    u64 wrr_runtime;
    /* Nests inside the rq lock: */
    raw_spinlock_t wrr_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
    unsigned long wrr_nr_boosted;

    struct rq *rq;
    struct list_head leaf_wrr_rq_list;
    struct task_group *tg;
#endif
};
```

Fig. 7. Definition of struct wrr-rq

```
struct rq {
    /* runqueue lock: */
    raw_spinlock_t lock;

    /*...
    unsigned long nr_running;
    #define CPU_LOAD_IDX_MAX 5
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    unsigned long last_load_update_tick;
#ifdef CONFIG_NO_HZ...
#endif
    int skip_clock_update;

    /* capture load from *all* tasks on this cpu: */
    struct load_weight load;
    unsigned long nr_load_updates;
    u64 nr_switches;

    struct cfs_rq cfs;
    struct rt_rq rt;
    struct wrr_rq wrr;

#ifdef CONFIG_FAIR_GROUP_SCHED...
#endif
#ifdef CONFIG_RT_GROUP_SCHED...
#endif

#ifdef CONFIG_WRR_GROUP_SCHED
    struct list_head leaf_wrr_rq_list;
#endif
```

Fig. 8. Adding variables to struct rq

4.3 Revision on /kernel/sched/core.c

When revising core.c, sched-fork() , sched-fork() , wake-up-new-task() , scheduler-tick() , rt-mutex-setprio() , setscheduler() , sched-setscheduler() , sched-init() should be revised according to RT and FAIR, and eventually WRR should behave similarly to them.

4.4 Revision on /kernel/sched/rt.c

In rt.c, the ".next" variable in rt-sched-class should be modified in order to make it point to wrr-sched-class.

4.5 /kernel/sched/Makefile

In the makefile, wrr.o should be added for compilation.

5 Testing Results

After debugging the compiling successfully, we get a new kernel with WRR. By loading it into Android, we can check the performance of it.

Before testing, a program to set the scheduler should be implemented. sched-getscheduler is used to set the scheduler of the process.

By typing ps -P grep processtest , we can get sufficient information for sched-getschedule . After executing sched-getschedule , we can get the information from kernel like figures below. Apparently, the target task has switched to WRR policy.

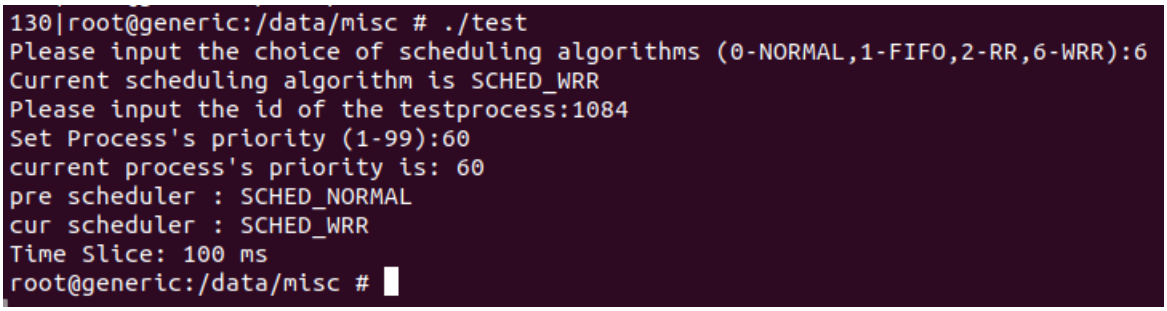


Fig. 9. Foreground process

We can see in the figure that the time slice of this process is 100 ms, meaning that it is in the foreground.

Then, by typing adb shell am start com.android.settings/.Settings, the process is put into the background. We test again and get the result shown in the following figure 10. This figure shows that the time

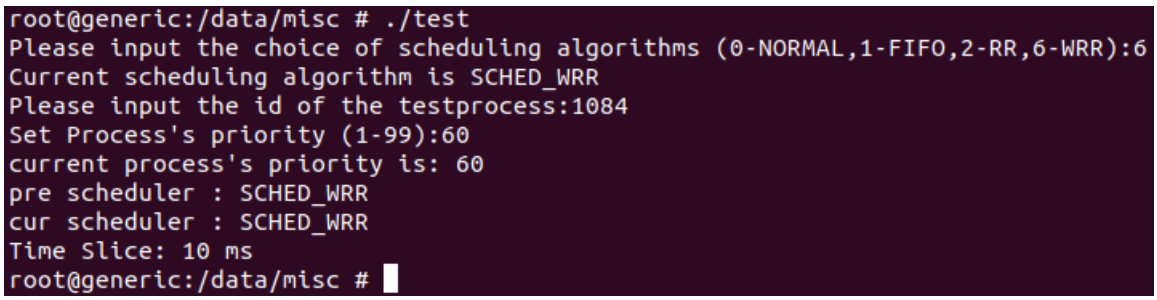


Fig. 10. Background process

slice of process in the background is 10 ms.

6 Extended Ideas

6.1 Comparison of Each Scheduler

A test program for the evaluation of the four common scheduling policies: NORMAL, FIFO, RR and WRR is implemented in this section.

In the testing process, we begin with setting the scheduling policy we want to teset. Then, a certain number of child processes (about 10) are forked from the original one. For each process, we do lots of simple additions to simulate the computation, and by recording the beginning time and ending time of the processes, we can compare the performance of each scheduler, which is shown in Table1. We can conclude

Scheduler	beginning time (ms)	ending time (ms)	total execution time (ms)
NORMAL	205022	268594	63572
FIFO	489393	516063	26670
RR	923765	950700	26935
WRR	315315	372813	57498

Table 1. Comparison of schedulers

that FIFO may be the best scheduling policy when it comes to the simple fork operations, which make sense because of its fewer context-switches.

6.2 Implementation of WRR with Priority

The WRR's assigning different time slices to processes in different states is somehow similar to the priority method, meaning that foreground processes have higher priority.

However, the time slice stays fixed until the state of the process changes, this feature may make WRR behave poorly in some certain situations. For example, a process is occupying the CPU for a long time, and the CPU should often reset the time slice, wasting lots of time.

Therefore, to solve this problem, we implement a new WRR with priority, which lowers the priority of the current task each time the time slice is reset. The lower the priority, the shorter the time slice, which may optimize the performance to some extent.