OS Project 2: Weight Round Robin Scheduler

517020910038, Peng Tang, 85704592@qq.com

June 5, 2021

Contents

1 Introduction

In this project, we are required to implement WRR scheduling policy to android core. WRR policy is used to assign tasks in foreground groups and background groups with different time slice. The tasks in foreground groups can get 100 milliseconds while tasks in background groups only have 10 milliseconds, which means tasks in foreground groups can running more time on cup. In order to implement WRR correctly, we need to modify the following files.

- /arch/arm/configs/goldfish_armv7_defconfig
- /include/linux/sched.h
- /kernel/sched/core.c
- /kernel/sched/sched.h
- /kernel/sched/rt.c
- /kernel/sched/Makefile
- /kernel/sched/debug.c
- /kernel/sched/wrr.c
- test_sched.c

The main work we need to do is write the program *wrr.c*, in which we defined all needed functions for WRR scheduler.

With above works, WRR policy is successfully implemented to the android core and we get the expected result by testing the new policy with the *processtest.apk* program.

2 Main Works

In this part, we will explain what we do in these files carefully.

2.1 /include/linux/sched.h

In this file, we decalre some necessary variables for WRR policy as follow

```
#define SCHED_WRR           6
```

This line is used to declare the WWR policy with 6, then we can add it to some need position.

```
struct wrr_rq;
```

This line is used to declare the WWR run queue structure.

```
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_WRR_GROUP_SCHED
        struct sched_wrr_entity *parent;

        struct wrr_rq           *wrr_rq;

        struct wrr_rq           *my_q;
#endif
};
```

This part is used to define the WRR entity, then we use it when scheduling task because it is unit of scheduling. The main members of this structure are *run_list* and *time_slice*. *run_list* is used to link the entity to the run queue, and *time_slice* is used to record the remaining time of the task.

```
#define WRRF_time slice          (100 * HZ /1000)
#define WRRB_time slice          (10 * HZ /1000)
```

This two lines are used to define the time slice for foreground tasks and background tasks.

```
struct task_struct {
.....
        struct sched_rt_entity rt;
        struct sched_wrr_entity wrr;
.....
};
```

This is used to add the WRR entity to the *task_struct* , then tasks can be scheduled through the WRR entity.

2.2 /kernel/sched/sched.h

In the file, we declare some variables and functions we need.

```
struct wrr_rq{
        struct list_head queue_wrr;
        unsigned long wrr_nr_running;

#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_WRR_GROUP_SCHED
```

```
        unsigned long wrr_nr_boosted;

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

struct rq {
....
        struct wrr_rq wrr;
.....
#ifdef CONFIG_WRR_GROUP_SCHED
        struct list_head leaf_wrr_rq_list;
#endif
....
};
```

The part is used to define the WRR run queue, which will be add to the *struct rq* showing above, then cpu can select task in *wrr_rq*.
The main member in *wrr_rq* is *queue_wrr*, which is used to link all WRR tasks by linking the WRR entity.

```
extern const struct sched_class wrr_sched_class;
....
extern void init_sched_wrr_class(void);
....
extern void init_wrr_rq(struct wrr_rq *wrr_rq, struct rq *rq);
```

The first line is used to declare the WRR class, which includes all the needed functions for the WRR policy.
The second line and third line are initializing functions, which are used to initialize the WRR class and WRR run queue.
All of them are defined in the *wrr.c* file.

2.3 /kernel/sched/core.c

This file is the main file for the android core to schedule tasks, so we need add some information about WRR to it, then we can implement the WRR policy.

```
static void __sched_fork(struct task_struct *p)
{
.....
        INIT_LIST_HEAD(&p->rt.run_list);
        INIT_LIST_HEAD(&p->wrr.run_list);
.....
}
```

In the *__sched_fork()* function, we add the *INIT_LIST_HEAD(&p- wrr.run_list)* to initialize the WRR entity's member *run_list*, which means we let the *run_list* points to itself.

```
static void
__setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)
```

```
{
.....
        if(policy == SCHED_WRR)
                p->sched_class = &wrr_sched_class;
.....
}
```

In the *___setscheduler()* function, we need to judge the policy is WRR whether or not when setting the policy for some task, if it is, then we need let the task's class be WRR class, so we add the two lines above in the function.

```
void __init sched_init(void)
{
.....
init_wrr_rq(&rq->wrr, rq);
.....
}
```

In the *___init sched_init(void)* function, we add *init_wrr_rq(& rq->wrr,rq)* to call the *init_wrr_rq()* function to initialize the WRR run queue.

```
static void free_sched_group(struct task_group *tg)
{
.....
        free_wrr_sched_group(tg);
.....
}
```

This is used to free the memory of the task group.

```
struct task_group *sched_create_group(struct task_group *parent)
{
.....
        if (!alloc_wrr_sched_group(tg, parent))  //
                goto err;
.....
}
```

This is used to call the *alloc_wrr_sched_group(struct task_group *tg, struct task_group *parent)* function to malloc the memory for the WRR run queue.

2.4 /kernel/sched/rt.c

```
const struct sched_class rt_sched_class = {
        .next                   = &wrr_sched_class,

        .....
};
```

In this file, we need to change the RT class's next class to be WRR class because WRR class have the higher priority than FAIR class.

2.5 /kernel/sched/wrr.c

This program is main work we need to do. In this file, we define many functions for the WRR policy, here we just analyse some of them.

```
const struct sched_class wrr_sched_class = {
        .next              =&fair_sched_class,
        .enqueue_task      =enqueue_task_wrr,
        .dequeue_task      =dequeue_task_wrr,
        .yield_task        =yield_task_wrr,
        .check_preempt_curr =check_preempt_curr_wrr,
        .pick_next_task           =pick_next_task_wrr,
        .put_prev_task            =put_prev_task_wrr,
        .task_fork                =task_fork_wrr,

#ifdef CONFIG_SMP
        .select_task_rq =select_task_rq_wrr,
        .set_cpus_allowed         =set_cpus_allowed_wrr,
        .rq_online                =rq_online_wrr,
        .rq_offline               =rq_offline_wrr,
        .pre_schedule             =pre_schedule_wrr,
        .post_schedule            =post_schedule_wrr,
        .task_woken               =task_woken_wrr,
#endif
        .switched_from            =switch_from_wrr,
        .set_curr_task            =set_curr_task_wrr,
        .task_tick                =task_tick_wrr,
        .get_rr_interval          =get_rr_interval_wrr,
        .prio_changed             =prio_changed_wrr,
        .switched_to              =switched_to_wrr,

};
```

The WRR scheduled class structure contains all functions we need to define. And the next scheduled class after WRR scheduled class is Fair scheduled class, so we need add *.next =&fair_sched_class* in this structure.

```
static void __enqueue_wrr_entity(struct sched_wrr_entity *wrr_se, bool head)
{
        struct wrr_rq *wrr_rq = wrr_rq_of_se(wrr_se);
        struct list_head *queue=&wrr_rq->queue_wrr;
        struct wrr_rq *group_rq = group_wrr_rq(wrr_se);
        struct task_struct *p=wrr_task_of(wrr_se);

        if (group_rq && (wrr_rq_throttled(group_rq) || !group_rq->wrr_nr_running))
                return;

        if (!wrr_rq->wrr_nr_running) //add the wrr_rq to the rq
                list_add_leaf_wrr_rq(wrr_rq);

        //give the new running time
        struct task_group *taskgroup = p->sched_task_group;
      char group_path[1024];
     if (!(autogroup_path(taskgroup, group_path, 1024)))
```

```
    {
        if (!taskgroup->css.cgroup) {
            group_path[0] = '\0';
        }
        cgroup_path(taskgroup->css.cgroup, group_path, 1024);
    }

    if(group_path[1]=='b')
    {
        p->wrr.time_slice = WRRB_time slice;

        printk("Set the time slice for backgroup\n");
    }
    else if(group_path[1]!='b')
    {
        p->wrr.time_slice = WRRF_time slice;
        printk("Set the time slice for foregroup\n");
    }



        if (head)
                list_add(&wrr_se->run_list, queue);
        else
                list_add_tail(&wrr_se->run_list, queue);

        inc_wrr_tasks(wrr_se, wrr_rq);
}
```

In the WRR scheduled class, the *enqueue_task_wrr()* function is used to put a WRR task into the WRR run queue. It realises this by calling the *enqueue_wrr_entity()* function, then the *enqueue_wrr_entity()* function calls the *___enqueue_wrr_entity()* function to put the task into the WRR run queue.

In the *___enqueue_wrr_entity()* function, we need to judge the task belonging to which group, foreground groups or background groups, by calling the *autogroup_path()* function. After knowing the group, we assign corresponding time slice to the task. Then, we put the task into the WRR run queue by linking the *run_list* member in the task's WRR entity to the list queue. After doing this, the task is successfully put into the WRR run queue.

```
static void ___dequeue_wrr_entity(struct sched_wrr_entity *wrr_se)
{
        struct wrr_rq *wrr_rq = wrr_rq_of_se(wrr_se);
        struct list_head *head  = &wrr_rq->queue_wrr;

        list_del_init(&wrr_se->run_list);

        dec_wrr_tasks(wrr_se, wrr_rq);
        if (!wrr_rq->wrr_nr_running)
                list_del_leaf_wrr_rq(wrr_rq);
}
```

In the WRR scheduled class, the *dequeue_task_wrr()* function is used to pull out some task from the run queue. It realises this by calling the *dequeue_wrr_entity()* function, then the *dequeue_wrr_entity()* function calls the *___dequeue_wrr_entity()* function to do this work.

In the *___dequeue_wrr_entity()* function, we just need to use the *list_del_init()* function to delete the task from the run queue list.

```
static struct task_struct *_pick_next_task_wrr(struct rq *rq)
{
        struct sched_wrr_entity *wrr_se;
        struct task_struct *p;
        struct wrr_rq *wrr_rq;

        wrr_rq = &rq->wrr;

        if (!wrr_rq->wrr_nr_running)
                return NULL;

        if (wrr_rq_throttled(wrr_rq))
                return NULL;

        do {
                wrr_se = pick_next_wrr_entity(rq, wrr_rq);
                BUG_ON(!wrr_se);
                wrr_rq = group_wrr_rq(wrr_se);
        } while (wrr_rq);

        p = wrr_task_of(wrr_se);
        p->se.exec_start= rq->clock_task;

        return p;
}
static struct sched_wrr_entity *pick_next_wrr_entity(struct rq *rq,
                struct wrr_rq *wrr_rq)
{
        struct sched_wrr_entity *next = NULL;
        struct list_head *queue=&wrr_rq->queue_wrr;
        next = list_entry(queue->next, struct sched_wrr_entity, run_list);
        return next;
}
```

In the WRR scheduled class, the *pick_next_task_wrr()* function is used to pick the next task in the WRR run queue to run on cpu. It realises this by calling the *_pick_next_task_wrr()* function, then *_pick_next_task_wrr()* function calls the *pick_next_wrr_entity()* function to find the next task in the run queue.

```
static void update_curr_wrr(struct rq *rq)
{
        struct task_struct *curr = rq->curr;
        struct sched_wrr_entity *wrr_se = &curr->wrr;
        struct wrr_rq *wrr_rq = wrr_rq_of_se(wrr_se);
        u64 delta_exec;

        if (curr->sched_class != &wrr_sched_class)
                return;
```

```
	delta_exec = rq->clock_task - curr->se.exec_start;
	if (unlikely((s64)delta_exec < 0))
		delta_exec = 0;

	schedstat_set(curr->se.statistics.exec_max,
		      max(curr->se.statistics.exec_max, delta_exec));

	curr->se.sum_exec_runtime += delta_exec;
	account_group_exec_runtime(curr, delta_exec);

	curr->se.exec_start = rq->clock_task;
	cpuacct_charge(curr, delta_exec);

	sched_wrr_avg_update(rq, delta_exec);



	for_each_sched_wrr_entity(wrr_se) {
		wrr_rq = wrr_rq_of_se(wrr_se);

		if (sched_wrr_runtime(wrr_rq) != RUNTIME_INF) {
			raw_spin_lock(&wrr_rq->wrr_runtime_lock);
			wrr_rq->wrr_time += delta_exec;
			if (sched_wrr_runtime_exceeded(wrr_rq))
				resched_task(curr);
			raw_spin_unlock(&wrr_rq->wrr_runtime_lock);

		}
	}
}
```

This function is not defined in the WRR scheduled class, but many functions in the class need to call it to update the running time of the task, if the task exceeds the limit time, the function will call the *resched_task()* function to pull out the task from the cpu.

```
static void task_tick_wrr(struct rq *rq, struct task_struct *p, int queued)
{
	struct sched_wrr_entity *wrr_se = &p->wrr;

	update_curr_wrr(rq);

	watchdog(rq, p);

	/*
	 * RR tasks need a special form of time slice management.
	 * FIFO tasks have no time slices.
	 */
	if (p->policy != SCHED_WRR)
		return;

	if (--p->wrr.time_slice)  //the time is not used up
		return;

	struct task_group *taskgroup = p->sched_task_group;
```

```
        char group_path[1024];
       if (!(autogroup_path(taskgroup, group_path, 1024)))
      {
          if (!taskgroup->css.cgroup) {
              group_path[0] = '\0';
          }
          cgroup_path(taskgroup->css.cgroup, group_path, 1024);
      }

      if(group_path[1]=='b')
      {
          p->wrr.time_slice = WRRB_time slice;

          printk("set the time slice for backgroup\n");
      }
      else if(group_path[1]!='b')
      {
          p->wrr.time_slice = WRRF_time slice;

          printk("set the time slice for foregroup\n");
      }


          /*
           * Requeue to the end of queue if we (and all of our ancestors) are the
           * only element on the queue
           */
          for_each_sched_wrr_entity(wrr_se) {
                  if (wrr_se->run_list.prev != wrr_se->run_list.next) {
                          requeue_task_wrr(rq, p, 0);
                          set_tsk_need_resched(p);
                          return;
                  }
          }
}
```

This function will be called by the *scheduler_tick* function in the *core.c* to update(by calling the *update_curr_wrr()* function) and check the WRR task's state, if it find some WRR task used up its time slice, then it will assign the new time slice to the task and put it into the WRR run queue again by calling the *requeue_task_wrr()* function.

By defining these functions of the WRR scheduler correctly, we can realise the WRR policy to reach the excepted result.

3 WRR Executing

1 Setting policy steps

The WRR scheduler works as the following steps.

- When we use the *sched_setscheduler()* syscall to set the scheduling for same task, this syscall will call the *do_sched_setscheduler(pid_t pid, int policy, struct sched_param __user *param)* function.

- Then the *do_sched_setscheduler(pid_t pid, int policy, struct sched_param __user *param)* functionn will find the task's *task_struct* through the pid we give, then it will call the *sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param)* function.
- The *sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param)* function will call the *__sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param, bool user).*
- The *__sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param, bool user)* will call the *__setscheduler(struct rq *rq, struct task_struct *p, int policy, int prio)* to set the scheduled class for the task according the *policy* parameter we give(here the policy is 6). After setting the scheduled class(wrr_sched_class for us), the *__sched_setscheduler(struct task_struct *p, int policy, const struct sched_param *param, bool user)* function will call the *enqueue_task(struct rq *rq, struct task_struct *p, int flags)* function.
- the *enqueue_task(struct rq *rq, struct task_struct *p, int flags)* function will call the the *enqueue_task_wrr(struct rq *rq, struct task_struct *p, int flags)* in the WRR scheduled class, then this task will be assigned the corresponding time slice and put into the WRR run queue.

With above steps, the task policy is set to the WRR policy.

2 Scheduling steps
When there is no real time task, the task in the WRR run queue will be scheduled to run on cpu.

- The *__sched __schedule(void)* will call the *pick_next_task(struct rq *rq)* function to pick the next task.
- The *pick_next_task(struct rq *rq)* function will traverse the scheduled class by the defined order. When there are no tasks in the RT run queue, it will call the **pick_next_task_wrr(struct rq *rq)* function in the WRR scheduled class.
- The **pick_next_task_wrr(struct rq *rq)* function will pick a WRR task from the WRR run queue and return it to the *__sched __schedule(void)*, then this task will run on cpu.
- The *scheduler_tick(void)* function will call the *task_tick_wrr(struct rq *rq, struct task_struct *p, int queued)* periodically when there WRR task run on the cup.
- Once the task used up its time slice, the *task_tick_wrr(struct rq *rq, struct task_struct *p, int queued)* will assign new time slice to the task and put it into the run queue again, and call the *set_tsk_need_resched(struct task_struct *p)* to remind the core scheduler to pick new task.

With above steps, the WRR task successfully runs on cpu for once time.

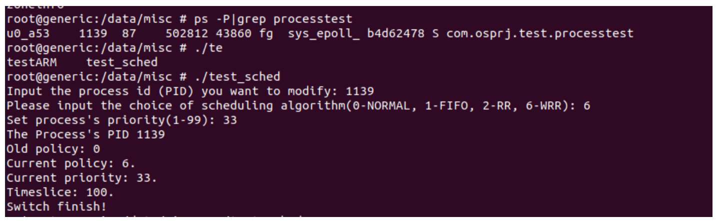
4 Testing Result

In order to test the WRR scheduler, we write the *test_sched.c* program to change the task's scheduling policy.
In the *test_sched.c* program, we use the following system call to set the scheduling policy for tasks.

```
sched_setscheduler(pid, policy, &param);
sched_getscheduler(pid);
sched_rr_get_interval(pid, &time_slice);
sched_getparam(pid, &param1);
```

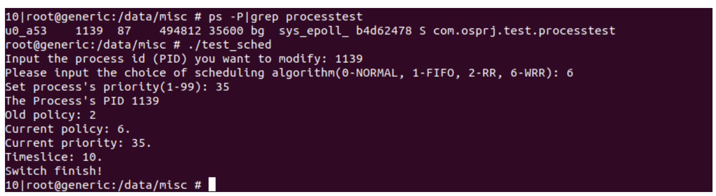
The first system call is used to set policy for the task through the pid.
The second system call is used to get the current policy of the task through the pid.
The third system call is used to get the timesilce of the task through the pid.
The fourth system call is used to get the priority of the task through the pid.

Through the *test_sched.c* program, we test the *processtest.apk* process and obtain the following result.



```
root@generic:/data/misc # ps -P|grep processtest
u0_a53    1139  87    502812 43860 fg  sys_epoll_ b4d62478 S com.osprj.test.processtest
root@generic:/data/misc # ./te
testARM    test_sched
root@generic:/data/misc # ./test_sched
Input the process id (PID) you want to modify: 1139
Please input the choice of scheduling algorithm(0-NORMAL, 1-FIFO, 2-RR, 6-WRR): 6
Set process's priority(1-99): 33
The Process's PID 1139
Old policy: 0
Current policy: 6.
Current priority: 33.
Timeslice: 100.
Switch finish!
```

Figure 1: Set the WWR policy for the task in foreground group



```
10|root@generic:/data/misc # ps -P|grep processtest
u0_a53    1139  87    494812 35600 bg  sys_epoll_ b4d62478 S com.osprj.test.processtest
root@generic:/data/misc # ./test_sched
Input the process id (PID) you want to modify: 1139
Please input the choice of scheduling algorithm(0-NORMAL, 1-FIFO, 2-RR, 6-WRR): 6
Set process's priority(1-99): 35
The Process's PID 1139
Old policy: 2
Current policy: 6.
Current priority: 35.
Timeslice: 10.
Switch finish!
10|root@generic:/data/misc #
```

Figure 2: Set the WWR policy for the task in background group

Form figure 1 and figure 2, we can see the result is what we want. For the foreground group, the time slice is 100 milliseconds; for the background group, the time slice is 10 milliseconds. And in the two cases, their policy is set to be the WRR policy.

5 Further Work

We write a test program to compare the performance of WRR, RR, FIFO, and NORMAL. With setting different policy for this program, we can get different executing time through the *time* command. The command format is like

- *time ./compare 0*
- *time ./compare 1*
- *time ./compare 2*
- *time ./compare 6*

Through the command above, we get results shown in figure 3, figure 4, figure 5, and figure 6. From these results, we can see the executing time of NORMAL is much shorter than others, and the the executing time of WRR, RR, and FIFO is almost same. We think this is because the time to pick next task for NORMAL policy is shorter while the other three is longer and their methods to pick next task are similar.

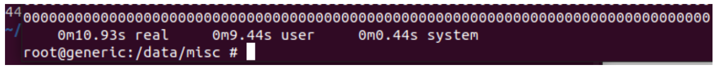


Figure 3: The executing time of NORMAL

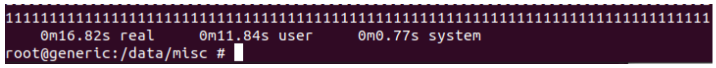


Figure 4: The executing time of FIFO

6 Conclusion

In this project, we add a new scheduling policy(WRR policy) to android core to assign different time slice for foreground group and background group. By testing WRR policy with the *processtest.apk*, we obtain the excepted result. Thus, we complete the project successfully.

6.1 Problems and Difficulties

When doing this project, we meet many problems and difficulties. The following problems are the main difficulties.

- The first difficulty we meet is to read and understand the meaning of these function in the *core.c* and *rt.c*. Because there are so many functions in these file, we don't know how to start. Luckily, by reading the blog provided in the slide, we begin to know what we need to do.
- When writing the *wrr.c* program, because this program uses so many pointers, it becomes more difficult to make sure the correctness.
- We can't understand the group scheduling and the SMP in the core.

6.2 Feeling

Through this project, we begin to understand the scheduling algorithm how to apply in the android core. And we know the steps about context exchange after reading so many kernel codes. Although we complete the WRR policy successfully, there are many things we can't understand. For example, we can't understand the group scheduling clearly and don't know how tasks are allocated to different cpus.

In conclusion, the task scheduling in android core is very difficult for us to understand it fully.

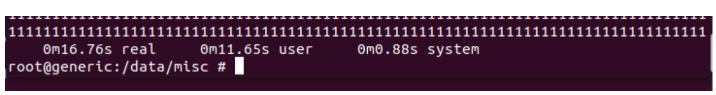


Figure 5: The executing time of RR

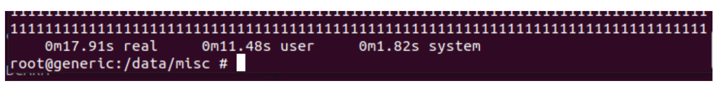


Figure 6: The executing time of WRR