Android Kernel Scheduler

1. Overview

```
├── Scheduler
|   ├── arch
|   |   └── goldfish_armv7_defconfig
|   ├── linux
|   |   └── sched.h
|   ├── sched
|   |   ├── core.c
|   |   ├── Makefile
|   |   ├── rt.c
|   |   ├── sched.h
|   |   └── wrr.c
|   └── test
|       ├── set_sched
|       |   ├── jni
|       |   |   ├── Android.mk
|       |   |   └── set_sched.c
|       |   ├── libs
|       |   |   └── armeabi
|       |   |       └── set_sched
|       |   └── obj
|       |       └── local
|       |           └── armeabi
|       |               ├── objs
|       |               |   └── set_sched
|       |               |       ├── set_sched.o
|       |               |       └── set_sched.o.d
|       |               └── set_sched
|       └── wrr_info
|           ├── jni
|           |   ├── Android.mk
|           |   └── wrr_info.c
|           ├── libs
|           |   └── armeabi
|           |       └── wrr_timeslice
|           └── obj
|               └── local
|                   └── armeabi
|                       ├── objs
|                       |   └── wrr_timeslice
|                       |       ├── wrr_info.o
|                       |       └── wrr_info.o.d
|                       └── wrr_timeslice
├── benchmark
|   ├── cpubound
|   |   ├── jni
|   |   |   ├── Android.mk
|   |   |   └── cpu_bound.c
|   |   ├── libs
|   |   |   └── armeabi
|   |   |       └── cpubound_test
```

```
|   |   └── obj
|   |       └── local
|   |           └── armeabi
|   |               ├── cpubound_test
|   |               └── objs
|   |                   └── cpubound_test
|   |                       ├── cpu_bound.o
|   |                       └── cpu_bound.o.d
|   ├── iobound
|   |   ├── jni
|   |   |   ├── Android.mk
|   |   |   └── io_bound.c
|   |   ├── libs
|   |   |   └── armeabi
|   |   |       └── iobound_test
|   |   └── obj
|   |       └── local
|   |           └── armeabi
|   |               ├── iobound_test
|   |               └── objs
|   |                   └── iobound_test
|   |                       ├── io_bound.o
|   |                       └── io_bound.o.d
|   └── mixbound
|       ├── jni
|       |   ├── Android.mk
|       |   └── mix_bound.c
|       ├── libs
|       |   └── armeabi
|       |       └── mixbound_test
|       └── obj
|           └── local
|               └── armeabi
|                   ├── mixbound_test
|                   └── objs
|                       └── mixbound_test
|                           ├── mix_bound.o
|                           └── mix_bound.o.d
└── OS project2 report.pdf

51 directories, 38 files
```

2. Instruction

2.1 Objectives

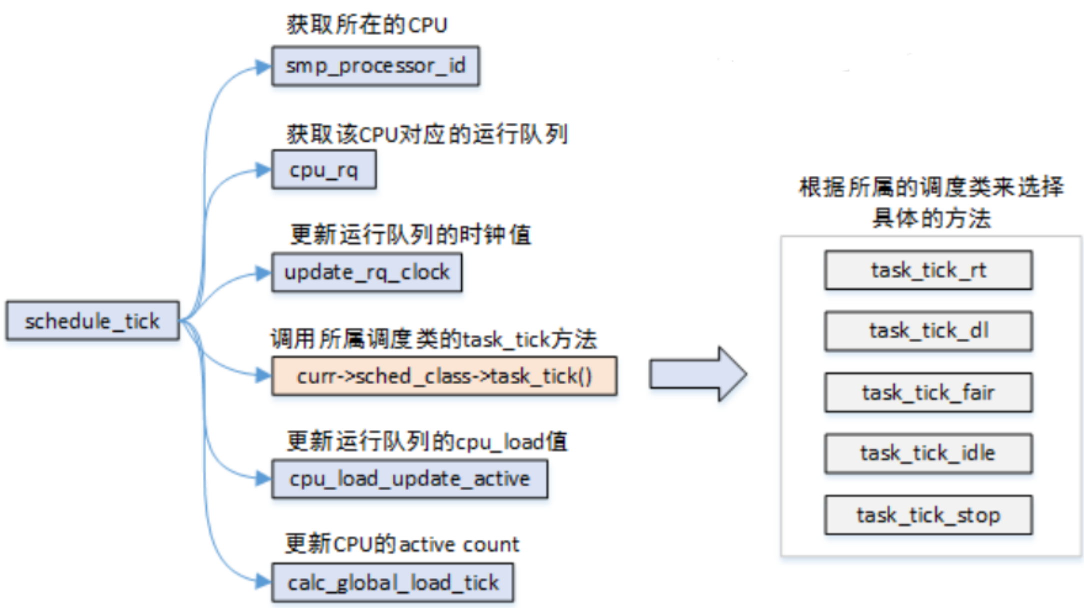
- Compile the Android kernel.
- Familiarize Android scheduler
- Implement a weight round robin scheduler
- Get experience with software engineering techniques.
- Bonus

2.2 Preliminary

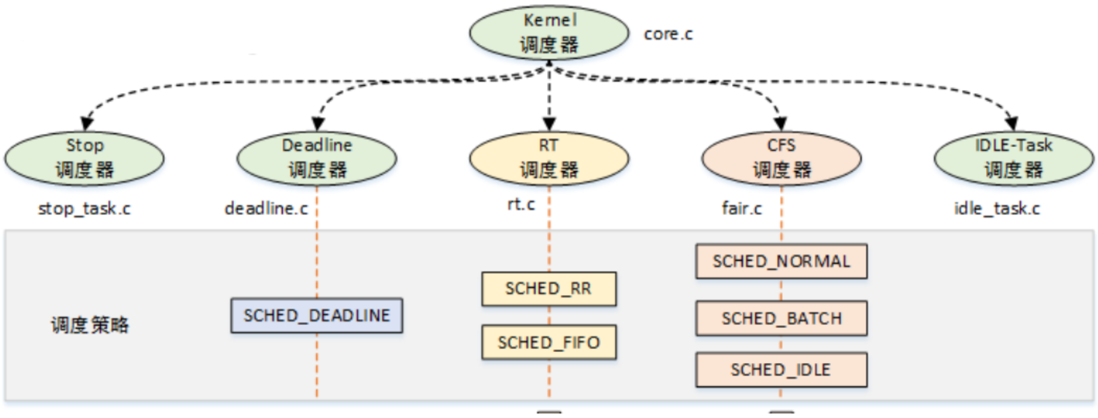
Process scheduling is one of the most important part of operating system. Before achieving our project, we need to understand how does the process scheduler work. I referred lots of material and website to familiarize the Android scheduler. Below is a brief instruction.

`schedule` is the main body of core scheduler, which is also called "the main scheduler function". The main scheduler is responsible for switching the CPU usage rights from one process to another process. To achieve the switching CPU usage, there are lots of work to do, such as put_prev_task, pick_next_task and context_switch.

`schedule_tick` : is periodically called by the kernel with the frequency `HZ`, which is the tick rate of the system timer defined on system boot. The first thing among what `scheduler_tick()` does is updating clocks invoking `update_rq_clock()`. The `update_rq_clock()` reads a clock source and updates the `clock` of the run queue, which the scheduler’s time accounting is based on. The second thing is checking if the current thread is running for too long, and if it is, setting a flag that indicates that `__schedule()` must be called to replace the running task with another. This done by calling `task_tick` in a scheduler class.

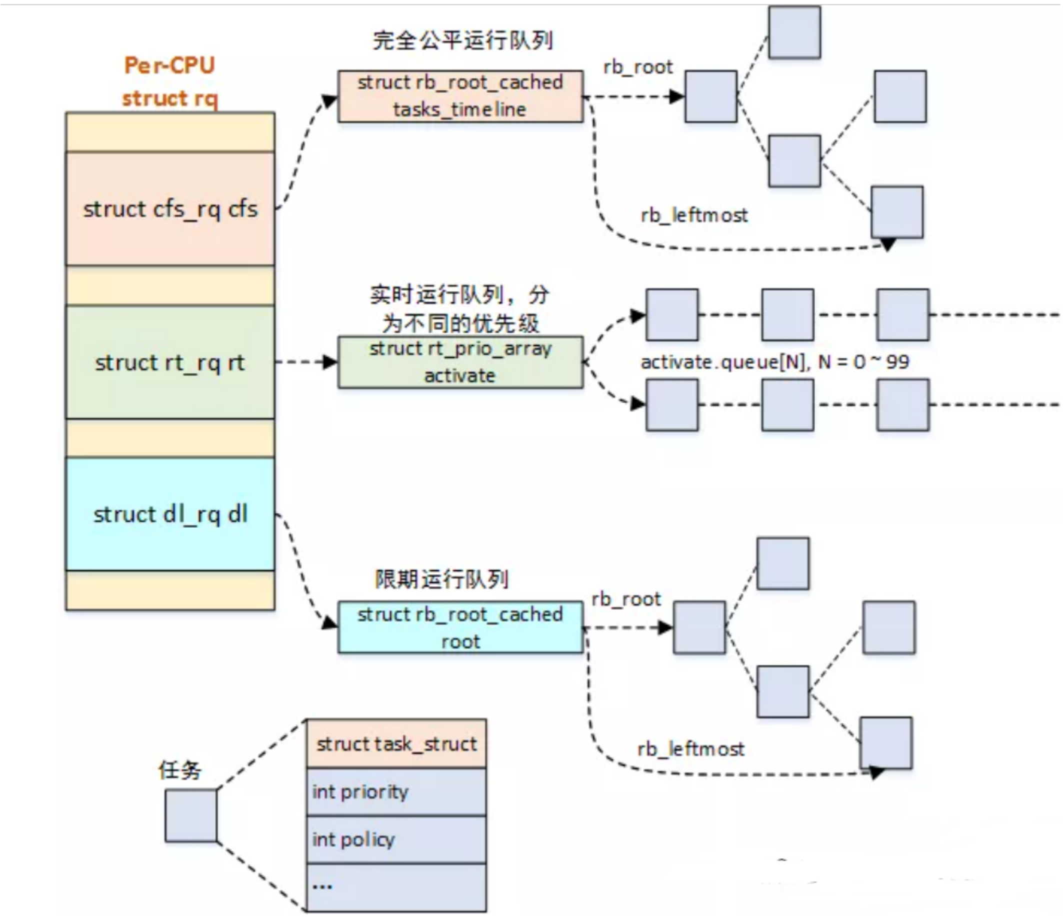


Scheduling class & Scheduling policy: There are five scheduler classes provided by the android kernel. Each of them support different schedule policy. For example, `sched_class_rt` supports FIFO(first in first out) and RR(round robin) policy. Each scheduling algorithm gets an instance of `struct sched_class` and connects the function pointers with their corresponding implementations.



The scheduling class we need to implement is also based on `struct sched_class`. As I understand, the scheduling class provides an set of API for the scheduling function to use. The major part of out project is implement the API in `wrr.c`.

Run queue & scheduling entity: Each CPU has a run queue, each scheduler acts on the run queue. In `struct rq`, there are some different run queue structs inserted into rq. Such as `struct rt_rq` and `struct cfs_rq`. Besides, the entry of each run queue is called scheduling entity, which is inserted in the PCB `struct task_struct`. Thus, for our project, we not only need to define the run queue `struct wrr_rq` and scheduling entity `struct sched_wrr_entity`, but also add to into `struct rq` and `struct task_struct`.



2.3 Major task

- I need to write codes in:
  - /kernel/sched/wrr.c
- I need to revise the following files to put wrr.c into effect
  - /arch/arm/configs/goldfish_armv7_defconfig
  - /include/linux/sched.h
  - /kernel/sched/sched.h
  - /kernel/sched/core.c
  - /kernel/sched/Makefile
- I need to write a test script to print the Scheduling Information of processtest.apk in foreground and background.

3. Implement

3.1 Modifications in other files

3.1.1 /arch/arm/configs/goldfish_armv7_defconfig

- **Configure WRR_GROUP_SCHED**

```
CONFIG_WRR_GROUP_SCHED=y
```

In this file, we enable the configuration of WRR group scheduling to activate wrr group scheduling code sections.

3.1.2 /include/linux/sched.h

- **Define SCHED_WRR**

```
#define SCHED_WRR       6
```

We define a new kind of schedule policy called SCHED_WRR. The value of SCHED_WRR should be 6.

- **Define sched_wrr_entity**

```
struct sched_wrr_entity {
    struct list_head run_list;  //调度实体在queue中的结点
    unsigned long timeout;
    int weight;                 //根据前后台所决定的权重
    unsigned int time_slice;    //调度实体当前所剩时间片
    int nr_cpus_allowed;

    struct sched_wrr_entity *back;

    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;

};
```

In the part, we define a struct called `sched_wrr_entity`, which will be operated as an entry of run queue. In this struct, we record the information that will be used for process scheduling. For example, by using `run_list` , we can link different entities into a list that forms a run queue (rq) for scheduler, besides, the element time_slice represent how many time slices dose this process remain.

- **Define time slice**

```
#define WRR_FG_TIMESLICE    (100 * HZ / 1000)
#define WRR_BG_TIMESLICE    (10 * HZ / 1000)
```

As the project required, we need to allocate different time slice to different process according to the group is foreground or background group. In our problem, we assign 100 ms for fore and 10 ms for back.

- **Add a sched_wrr_entity variable to task_struct**

```
struct sched_wrr_entity wrr;
```

This struct should be inserted into Process Control Block——task_struct.

- **Declare a wrr_rq struct**

```
struct wrr_rq;
```

3.1.3 /kernel/sched/sched.h

- **Declare a wrr_rq struct**

```
struct wrr_rq;
```

- **Define a new struct wrr_rq**

```
struct wrr_rq
{
    struct list_head queue;    //运行对列的头结点
    unsigned long wrr_nr_running;//运行队列中的节点个数
    unsigned long total_weight;//运行队列总权重

    struct list_head leaf_wrr_rq_list;
    struct task_group *tg;
    struct rq *rq;

};
```

This part is to define `wrr_rq` struct for WRR scheduling, where `wrr_nr_running` denotes the number of entities in `wrr_rq`. Meanwhile, it owns a pointer to the general run queue and an entity queue of itself. This struct organize the `struct sched_wrr_entity` in a queue, whose head element is `wrr_rq.queue`.

- **Add a wrr_rq variable to struct rq.**

```
struct wrr_rq wrr;

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

The `struct wrr_rq` should be added into the `struct rq`. Each CPU has a run queue, each scheduler acts on the run queue. In `struct rq`, there are some different run queue structs inserted into rq.

- **Declare some extern variables and functions.**

```
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);
```

3.1.4 /kernel/sched/core.c

- **Revise function `__sched_fork(struct task_struct *p)`**

```
  INIT_LIST_HEAD(&p->wrr.run_list);
```

The function `_sched_fork` perform scheduler related setup for a newly forked process p. We add this command to initialize the head element of wrr run queue.

- **Revise function** `__setscheduler(struct rq,struct task_struct,int,int)`

```
    if(policy==SCHED_WRR){
         p->sched_class = &wrr_sched_class;
     }
     else    if (rt_prio(p->prio))
         p->sched_class = &rt_sched_class;
     else
         p->sched_class = &fair_sched_class;
```

This function actually do priority and policy change. We need to judge whether the policy of p is SCHED_WRR or not. If the policy is SCHED_WRR, then set the sched_class pointer to wrr_sched_class.

- **Revise function** `__sched_setscheduler(struct task_struct, int, const struct sched_param , bool)`

```
if (policy < 0) {
        reset_on_fork = p->sched_reset_on_fork;
        policy = oldpolicy = p->policy;
    }
else {
        reset_on_fork = !!(policy & SCHED_RESET_ON_FORK);
        policy &= ~SCHED_RESET_ON_FORK;

        if (policy != SCHED_FIFO && policy != SCHED_RR &&
                policy != SCHED_NORMAL && policy != SCHED_BATCH &&
                policy != SCHED_IDLE && policy != SCHED_WRR)
            return -EINVAL;
    }
```

In this part, we add some checking properties for WRR scheduling in case some original exceptions.

- **Revise function** `free_sched_group(struct task_group *tg)`

```
free_wrr_sched_group(tg);
```

- **Revise function** `*sched_create_group(struct task_group *parent)`

```
if (!alloc_wrr_sched_group(tg, parent))
        goto err;
```

3.1.5 /kernel/sched/rt.c

```c
const struct sched_class rt_sched_class = {
     /* set the priority of wrr_sched_class next to rt_sched_class */
    .next           = &wrr_sched_class,
    .enqueue_task       = enqueue_task_rt,
    .dequeue_task       = dequeue_task_rt,
    .yield_task     = yield_task_rt,

    .check_preempt_curr = check_preempt_curr_rt,

    .pick_next_task     = pick_next_task_rt,
    .put_prev_task      = put_prev_task_rt,
......
```

Now that we added a new sched_class, we need to add `wrr_sched_class` to sched_class link list to make it work. But we only set `wrr_sched_class`'s next sched_class, so no sched_class points to `wrr_sched_class`. Therefore we need additional modification in rt.c file.

3.2 Work on wrr.c

According to the request, here are some function we need to implement.

```c
const struct sched_class wrr_sched_class = {
    .next           = &fair_sched_class,        /*Required*/
    .enqueue_task       = enqueue_task_wrr,     /*Required*/
    .dequeue_task       = dequeue_task_wrr,     /*Required*/
    .yield_task     = yield_task_wrr,           /*Required*/
    .check_preempt_curr = check_preempt_curr_wrr,       /*Required*/
    .pick_next_task     = pick_next_task_wrr,     /*Required*/
    .put_prev_task      = put_prev_task_wrr,      /*Required*/
    .task_fork          = task_fork_wrr,
#ifdef CONFIG_SMP
    .select_task_rq     = select_task_rq_wrr,           /*Never need impl */
    .set_cpus_allowed       = set_cpus_allowed_wrr,     /*Never need impl */
    .rq_online              = rq_online_wrr,            /*Never need impl */
    .rq_offline             = rq_offline_wrr,           /*Never need impl */
    .pre_schedule       = pre_schedule_wrr,             /*Never need impl */
    .post_schedule      = post_schedule_wrr,            /*Never need impl */
    .task_woken     = task_woken_wrr,                   /*Never need impl */
    .switched_from      = switched_from_wrr,            /*Never need impl */
#endif
    .set_curr_task          = set_curr_task_wrr,        /*Required*/
    .task_tick      = task_tick_wrr,                    /*Required*/
    .get_rr_interval    = get_rr_interval_wrr,
    .prio_changed       = prio_changed_wrr,         /*Never need impl */
    .switched_to        = switched_to_wrr,          /*Never need impl */
};
```

3.2.1 enqueue_wrr_entity

This function is used to add an `sched_wrr_entity` into the run queue `wrr_rq`. The major part of this function is `list_add` and `list_add_tail`. These two functions provided by kernel give us a good way to add an entry into a list.

Note: Before we add the entry, we need to assign the weight to entity according to the background or foreground.

3.2.2 enqueue_task_wrr

This function will be used once a process is at the ready state. The call trajectory of the functions is: `enqueue_task () -> enqueue_task_wrr ()`

- Call the function `enqueue_wrr_entity()`
- Update the information of run queue by using `inc_nr_running`.

3.2.3 dequeue_wrr_entity

This function is used to remove an `sched_wrr_entity` out of the run queue `wrr_rq`. The major part of this function is `list_del_init`. This function provided by kernel gives us a good way to delete an entry out of a list.

3.2.4 dequeue_task_wrr

This function will be used once a process is be blocked. The call trajectory of the functions is: `dequeue_task () -> dequeue_task_wrr ()`

- Update the information of current task.
- Call the function `dequeue_wrr_entity()`
- Update the information of run queue by using `dec_nr_running`.

Note: Before we delete a process from the run queue, we are supposed to update the information of the current task.

3.2.5 pick_next_task_wrr

This function is be used to pick the next ready task in `wrr_rq` to execute. The call trajectory of the functions is:

`schedule()->__schedule()->pick_next_task()->pick_next_task_wrr()`

- Make sure the run queue is not empty by checking `wrr_nr_running`.
- Then pick the first entry of the run queue `wrr_rq` by calling `list_first_entry()`

3.2.6 put_prev_task_wrr

This function put the task back to `wrr_rq`. The call trajectory of the function is: `put_prev_task()->put_prev_task_wrr()`

- Update the information of current task.
- Secondly, set the starting execution time of the task to 0.

3.2.7 yield_task_wrr

This function indicates that the current process proactive temporarily waives the execution. The call trajectory of the function is: `do_sched_yield()->yield_task_wrr()`

- Implement this function by calling `requeue_wrr_entity`

3.2.8 requeue_wrr_entity

This function is used to move the entry to the tail of the run queue.

- the major part of this function is `list_move` and `list_move_tail`.

3.2.9 get_rr_interval_wrr

This function returns the time slice that assigned to each process. The call trajectory of the function is `do_sched_rr_get_interval()->get_rr_interval_rt()`.

- Obtain the group information of the given task (foreground/background) using function `cgroup_path` and `task_group`.
- Secondly, according to the group information, decide the assigned time slice for the task.

3.2.10 task_tick_wrr

This function is periodically called by the kernel with the frequency `HZ`, which is the tick rate of the system timer defined on system boot. The call trajectory of the function is `scheduler_tick()->task_tick_rt()`.

- Update the current task's runtime statistics of using function `update_curr_wrr`.
- Check whether the scheduling policy is WRR, otherwise, exit the function.
- Reduce the time slice of the task by 1 and exit if the time slice is not equal to 0 yet. Otherwise, reset the time slice for the task according to the task's group information, remove the task from run queue and finally reschedule the task.

4. Testing Result

Now that we have a new kernel with WRR scheduler, we can load it to Android emulator to see how well it works.

To test our new kernel, I write two test files——`set_sched.c` and `wrr_info.c.`

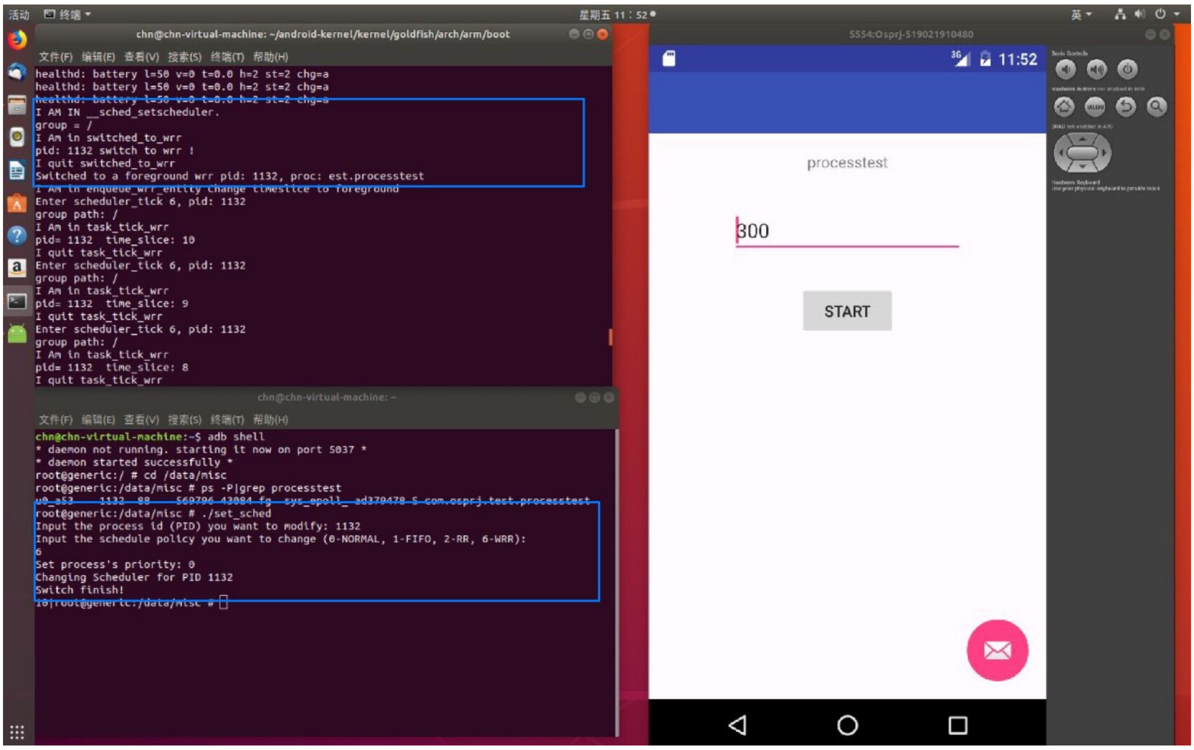
The first file is used to change the schedule policy. After we input some necessary information, this program will successfully change the policy by using function `set_sched`.

Note: If we change the policy to SCHED_WRR, then the priority is meaningless.

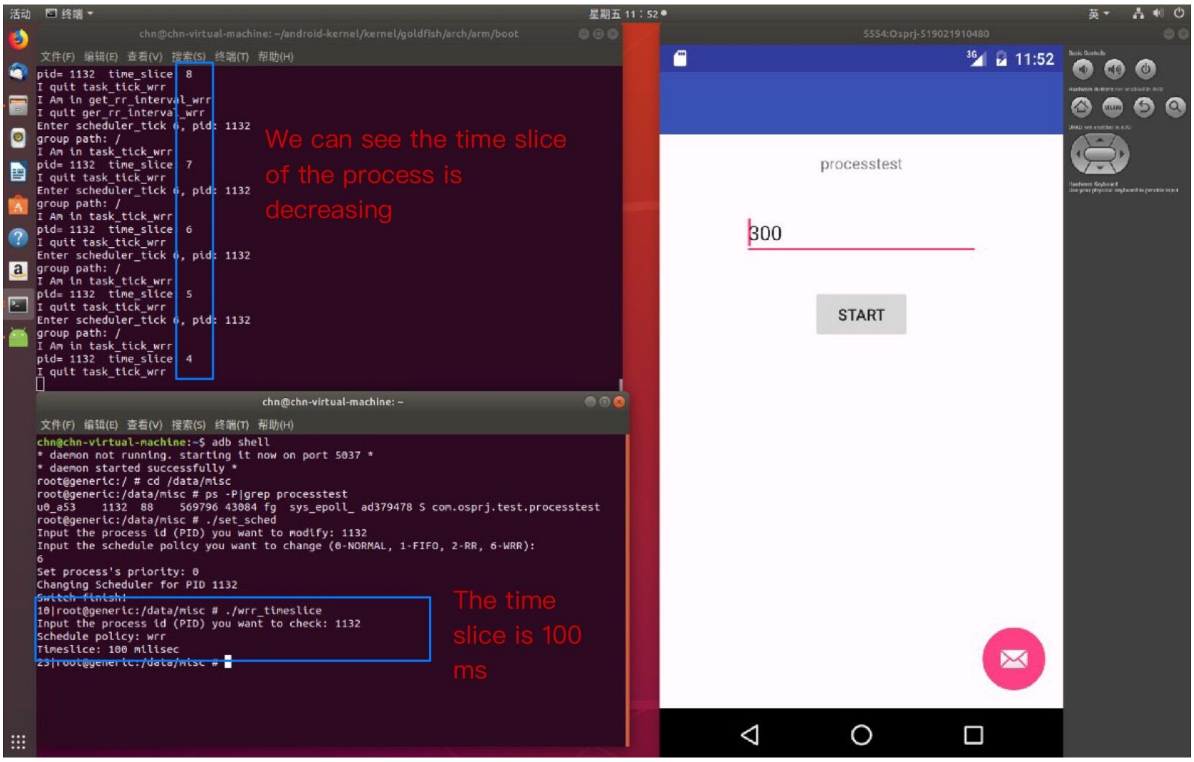
The second by using syscall `sched_getscheduler()` and `sched_rr_get_interval()` to a task's schedule policy information and execution time interval.

Following are some screen shot and some comments.

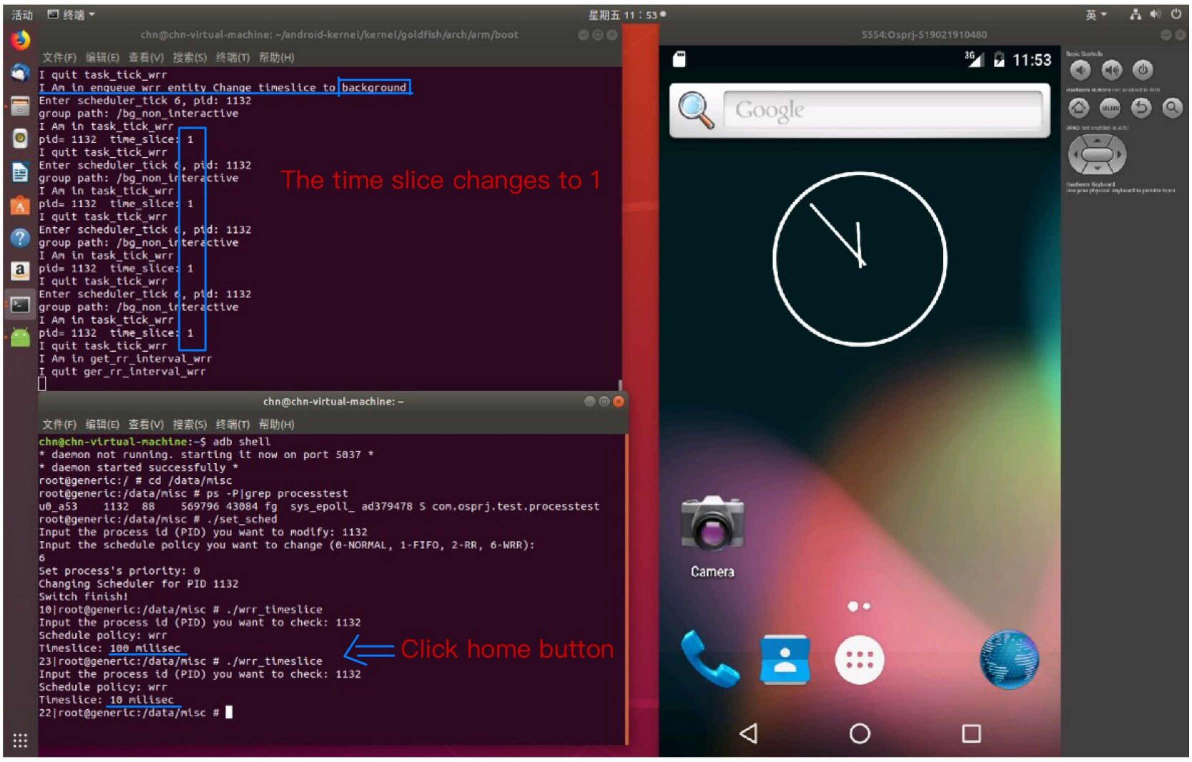
- Firstly, we used `ps -P|grep processtest` to get some information of the process, especially the pid of processtest. The we applied the test program to change the policy of processtest from normal to SHCED_WRR.



- Secondly, we used wrr_timeslice to check the timeslice of processtest. By the way, we can also get some information from the kernel terminal.



- Then we click home button in emulator, to see how kernel information changes. In figure below, we can see the state of this task changes to background and time slice also changes to background timeslice: 10ms.



5. Bonus: a method to compare the performance of schedule policy

5.1 Instruction

I come up with a method to compare the performance of RR, FIFO and WRR.

The basic idea is to use a kind of test program which is called benchmark.

To implement this idea, I write three kinds of benchmark—— `cpubound`, `iobound` and `mixbound`. Because I think maybe different types of process will effect the performance of different schedule policy.

- **I/O-bound vs. CPU-bound**

  Threads (or processes) can be classified into two major types: I/O-bound and CPU-bound.

  The I/O-bound threads are mostly waiting for arrivals of inputs or the completion of outputs. In general, these threads do not stay running for very long, and block themselves voluntarily to wait for I/O events.

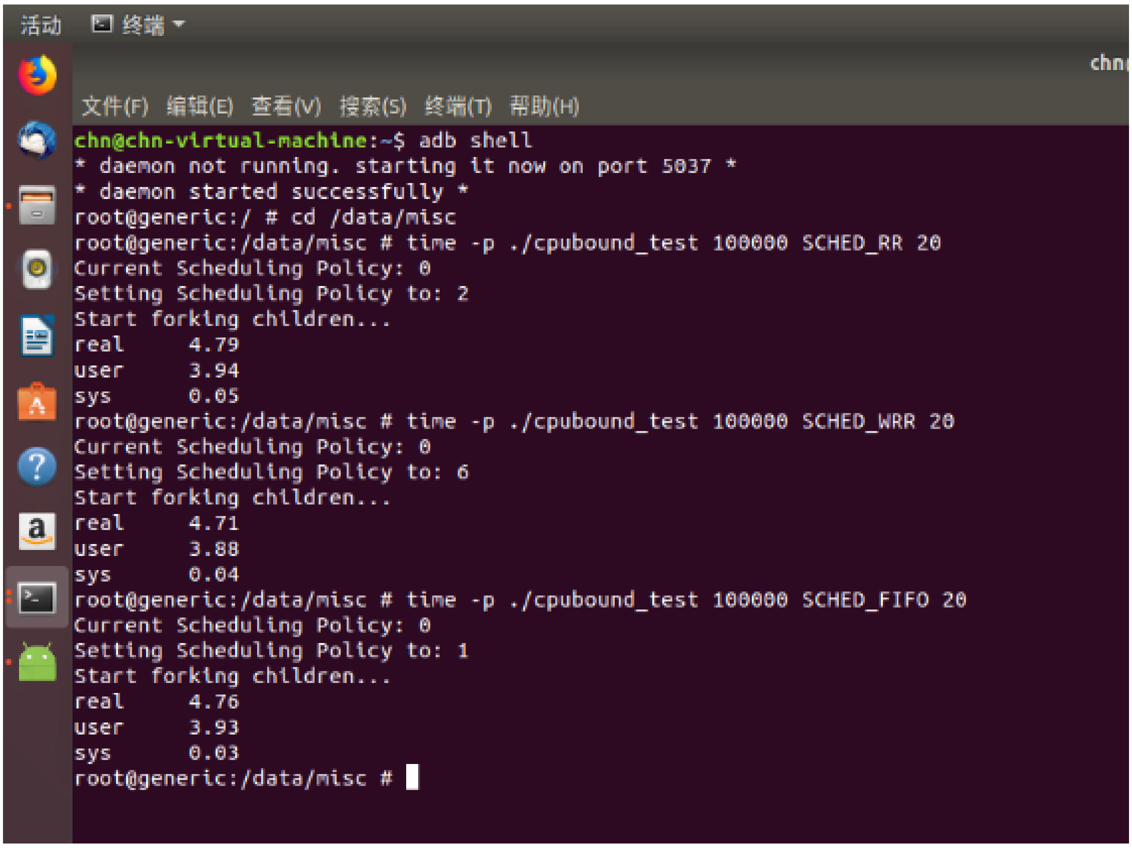
  The CPU-bound threads are ones that spend much of their time in doing calculations. Since there are not many I/O events involved, they tend to run as long as the scheduler allows. Typically, users do not expect the system to be responsive while the CPU-bound threads are running. Thus, the CPU-bound threads are picked to run by a scheduler less frequently.

5.2 cpubound_test

Benchmark for cpu-bound: In order to create a CPU-bound process, I write a child process which using the float number multiply, float number divide and function `sqrt,power` to calculate over and over again, which spend much of their time in doing calculations. To better control the scale of the program, we provide the interface for user to choose proper iteration times and forked processes to benchmark different scheduling policies through command-line parsing.

> command line: time -p ./cpubound_test Repeat_time Policy Fork_time

- **Result**:



5.3 iobound_test

Benchmark for io-bound: In order to create a IO-bound process, I write a child process which transfer the data from source file to destination file over and over again, which is mostly waiting for arrivals of inputs or the completion of outputs. To better control the scale of the program, we provide the interface for user to choose proper block size, transfer size and forked processes to benchmark different scheduling policies through command-line parsing.

> command line: time -p ./iobound_test Policy Source file Destination file Block_size Transfer_size Fork time

- **Result:**

```
ime -p ./iobound_test SCHED_FIFO ./data_in ./data_out 2000 5000000 20         <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 1
Starting forking children...
real      4.03
user      0.18
sys       3.13
_test SCHED_RR ./data_in ./data_out 2000 5000000 20                           <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 2
Starting forking children...
real      3.77
user      0.24
sys       2.93
nd_test SCHED_WRR ./data_in ./data_out 2000 5000000 20                        <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 6
Starting forking children...
real      3.84
user      0.23
sys       3.01
root@generic:/data/misc #
```

5.4 mixbound_test

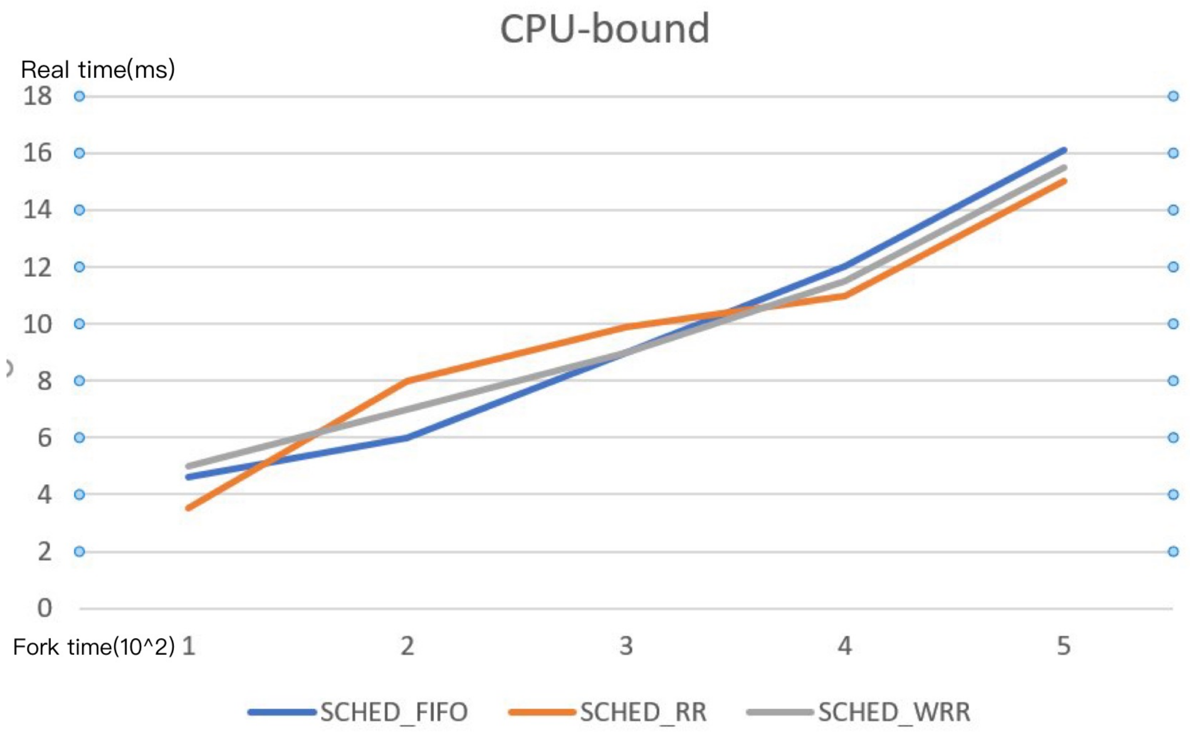
Benchmark for mix-bound: Combine both CPU-Bound and IO-Bound process.

- **Result:**

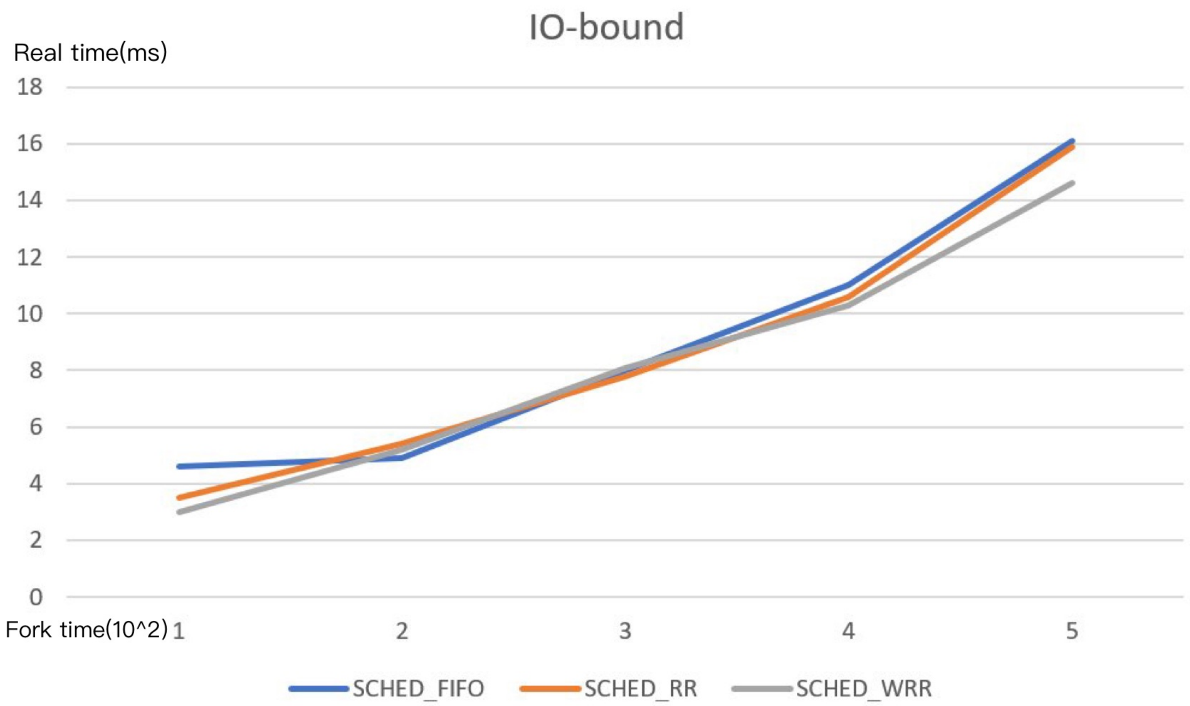
```
ime -p ./mixbound_test 100000 SCHED_RR 20 2000 500000 ./data_in ./data_out    <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 2
Start forking children...
real      5.68
user      4.11
sys       0.61
ime -p ./mixbound_test 100000 SCHED_WRR 20 2000 500000 ./data_in ./data_out   <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 6
Start forking children...
real      5.70
user      4.12
sys       0.59
 100000 SCHED_FIFO 20 2000 500000 ./data_in ./data_out                        <
Current Scheduling Policy: 0
Setting Scheduling Policy to: 1
Start forking children...
real      5.61
user      4.29
sys       0.51
root@generic:/data/misc #
```

5.6 Data process

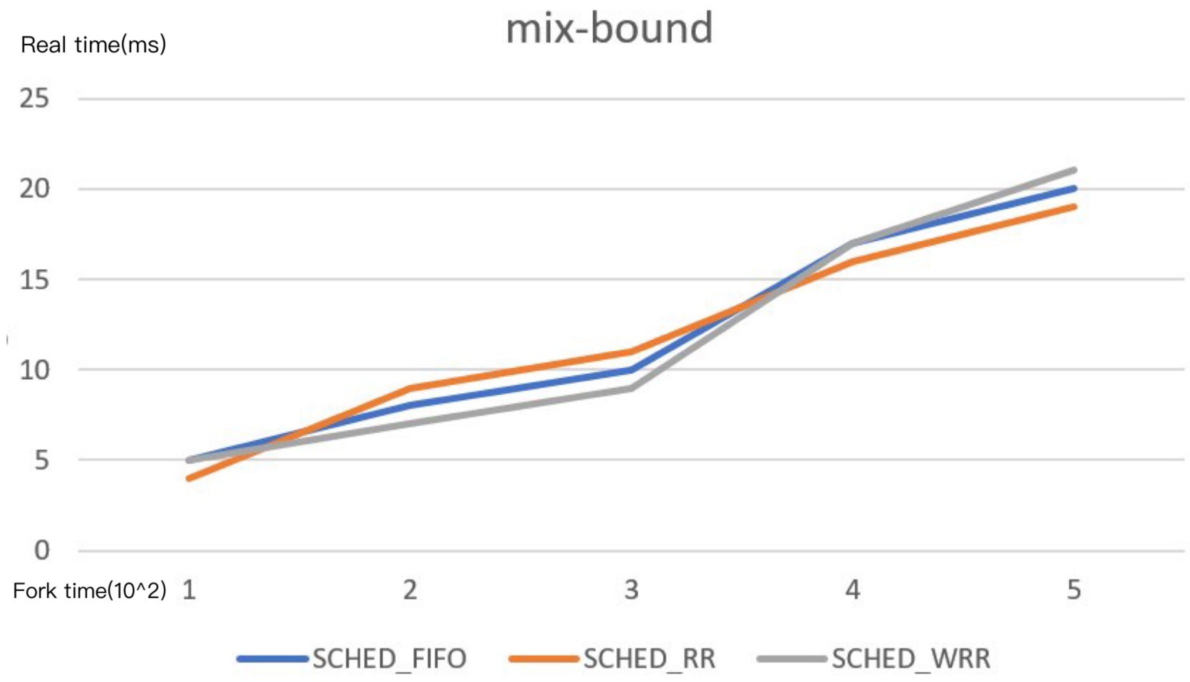
We select the data from the three kinds of benchmark. To better analysis the data, I create the below three graphs to visualize the performance of these three policies under a sequence of process numbers.



CPU-bound
Real time(ms)
18
16
14
12
10
8
6
4
2
0
Fork time(10^2) 1
2
3
4
5
SCHED_FIFO
SCHED_RR
SCHED_WRR



IO-bound
Real time(ms)
18
16
14
12
10
8
6
4
2
0
Fork time(10^2) 1
2
3
4
5
SCHED_FIFO
SCHED_RR
SCHED_WRR



- **Result: According to the graph above, I find out that in the IO-bound test, the SCHED_RR and SCHED_WRR performance is clearly better than SCHED_FIFO.**