OsPrj-2 Android Scheduler

report

name: Yizhen Guo

Student ID: 519021910507

Catalogue

1 wcc.c

1.1 Overview

Codes are mainly divided into a few parts: Pre-work、Enqueue、Dequeue、Yield and requeue、Pick Put、Task fork、Task tick、Get interval、set curr 、switch to. The framework of my codes is based on rt sched class and cfs sched class.

The wrr scheduler will judge the task before assigning time slice to it. Foreground task will get 100ms time slice and background task will get only 10ms.

1.2 Pre-work

Pack the simple functions that will be used later into the form that rt uses. For example:

```
static inline
void inc_wrr_tasks(struct sched_wrr_entity *wrr_se, struct wrr_rq *wrr_rq)
{
        wrr_rq->wrr_nr_running++;
}
static inline
void dec_wrr_tasks(struct sched_wrr_entity *wrr_se, struct wrr_rq *wrr_rq)
{
        WARN_ON(!wrr_rq->wrr_nr_running);
        wrr_rq->wrr_nr_running--;
}
```

it can help me coding faster and make the code easier to read for the people that is familiar with the rt scheduler.

1.3 Enqueue

The work of the part is much simpler than rt's enqueue because no priority is needed and therefore a single run queue can hold all the task entity. When enqueuing a new task, it first judge the flags to determine whether it should be added to the head of the list. Then increase the number of tasks on runqueue in both rq and wrr_rq. If it's flags is true and the ENQUEUEWAKEUP is true ,it mains it is a task currently waking up,so the timeout should be reset to 0.

1.4 Dequeue

Before dequeue, the rq should first update the curr task struct to increase total execute time of current task and use cpu accounting controller to charge the the time it consumes. After that, the task can be dequeued as an entity. Also decrease the number of tasks on runqueue in both rq and wrr_rq.

1.5 Yeild and requeue

As a weight round robin scheduler, yeilded task should be requeued to the end of the runqueue. So simple move it to the tail after checking it is an entity on the runqueue.

1.6 Pick and Put

Pick next task simply return the next task on wrr runqueue and set its execute start time to the current clock time. Put prev task simply re-execute the current task and reset its execute start time.

1.7 Task fork

The forked child process should run on the same cpu as its parent and also have the same time slice its parent has. To change the time slice need to update the rq clock so runqueue lock is neccessary as it is said in sched.h line 414. Code form reference is sched_fork() in core.c .

1.8 Get interval

This function simply return the time slice task should get.

1.9 Task tick

Every tick the running task should decrease the time slice it remains. If the time is used up it should be resched. Also use a watchdog to prevent deadlock and other risk, increase the timeout to see whether it takes too much time.

1.10 Set curr task

Simply set current task's start time to now.

1.11 Switch to wrr

Check its orignal scheduler and see whether it is on the runqueue. Only reschedule the task that wasn't in wrr runqueue.

1.12 Others

Declare the no implemented function and build the complete wrr sched class.

2 Other files

2.1 linux/sched.h 、sched/sched.h 、goldfish_armv7_defconfig

Same operation as in the slide.

2.2 core.c

1.Generally the same operation as in the slide but also add

```
void init_wrr_rq(struct wrr_rq *wrr_rq,struct rq *rq)
{
        INIT_LIST_HEAD(&wrr_rq->queue);
        wrr_rq->wrr_nr_running = 0;
```

```
}
```

to avoid redefination.

2.Added INIT_LIST_HEAD(p->wrr.run_list); in function:

static void ___sched_fork(struct task_struct *p);
also check the p->policy to set the sched_class as wrr_sched_class.

3.Add policy != SCHED_WRR in policy check and add

```
if ((rt_policy(policy) != (param->sched_priority != 0)) &&
    (wrr_policy(policy) != (param->sched_priority != 0)))
                return -EINVAL;
```

in function ___sched_setscheduer();

4.Add free_wrr_sched_group(tg); in function:

free_sched_group(struct task_group *tg)

5.In funtion: *sched_create_group(struct task_group *parent); added

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

6.Revise SYSCALL_DEFINE1 sched_get_priority_max() by added

```
case SCHED_WRR:
        ret = 99;
        break;
```

priority in wrr is useless, revise it just for convenience of the brenchmark operation.

7.Revise rt_sched_class

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

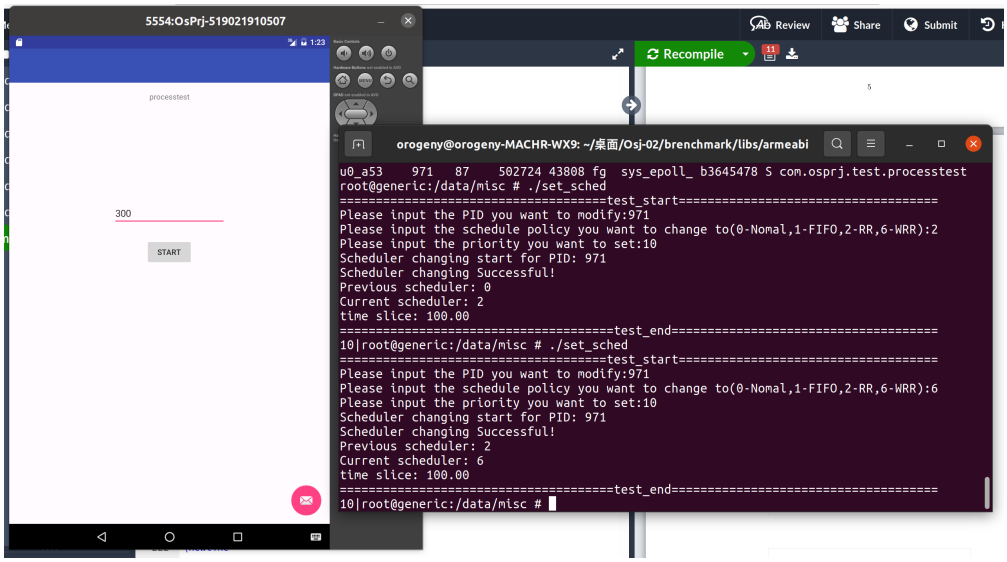
set the next scheduler class after rt to wrr. This will insert the wrr class between the rt and cfs class in the priority order, which means the scheduler core will try to finish all the task on the wrr runqueue after all tasks are done on the rt runqueue and before it turns to cfs class. It truly add the wrr scheduler into the core.

2.3 OUTPUT

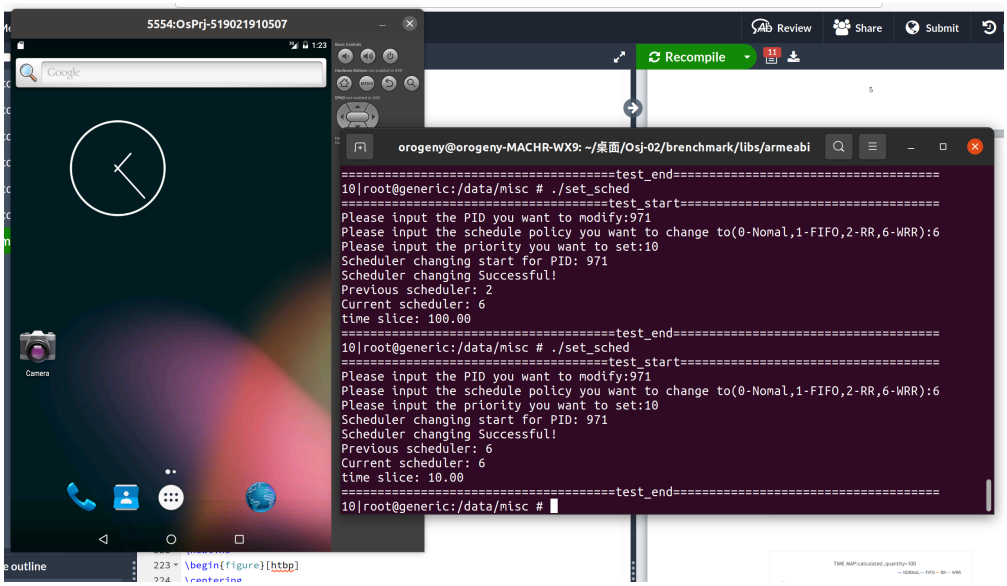
test output as follow:

```
====================================test_start====================================
Please input the PID you want to modify:971
Please input the schedule policy you want to change to(0-Nomal,1-FIFO,2-RR,6-WRR):2
Please input the priority you want to set:10
Scheduler changing start for PID: 971
Scheduler changing Successful!
Previous scheduler: 0
Current scheduler: 2
time slice: 100.00
====================================test_end======================================
```

Picture 1: switch to RR



Picture 2: switch to WRR(foreground)



Picture 3: switch to WRR(background)

3 brenchmark

Testing the performance of WRR, by comparing the total time cost of calculating some big random number using muiltiple child process. Use the command line ”time -p ./xxxx” to show the real time cost in different scheduler.

3.1 brenchmark code

The scheduler is set to the assigned policy and will fork ”num_childproc” child processes to run the following function:

```
long double calculation_task(int calculated_quantity){
    long double ans=1,total=0;
    long i;
    for(i = 0; i < calculated_quantity * 100; i++ ){
        ans= (random() / RAND_MAX) * (random() / RAND_MAX);
        total+= ans;
    }
    return total;
}
```

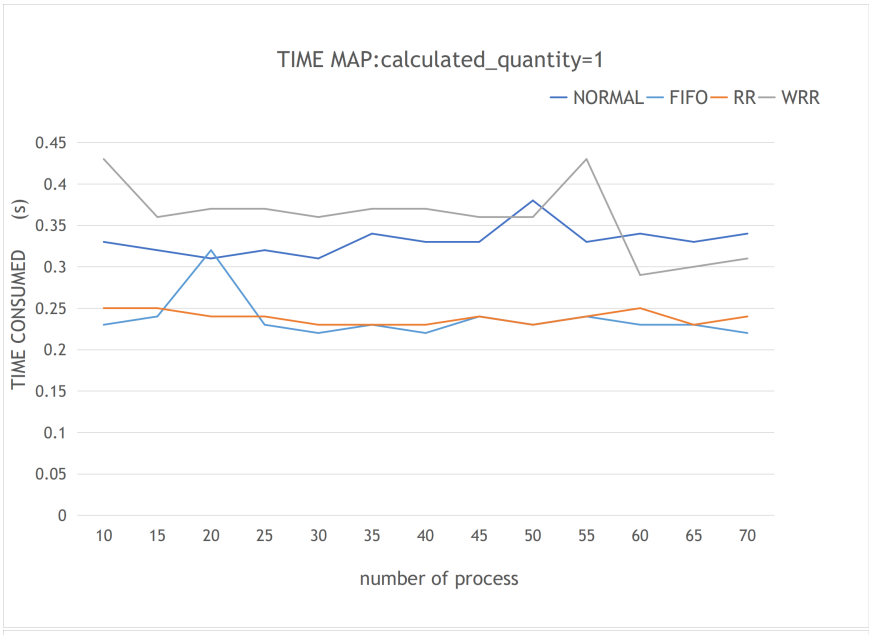
clearly that calculated_quantity determined the time needed for each child process.

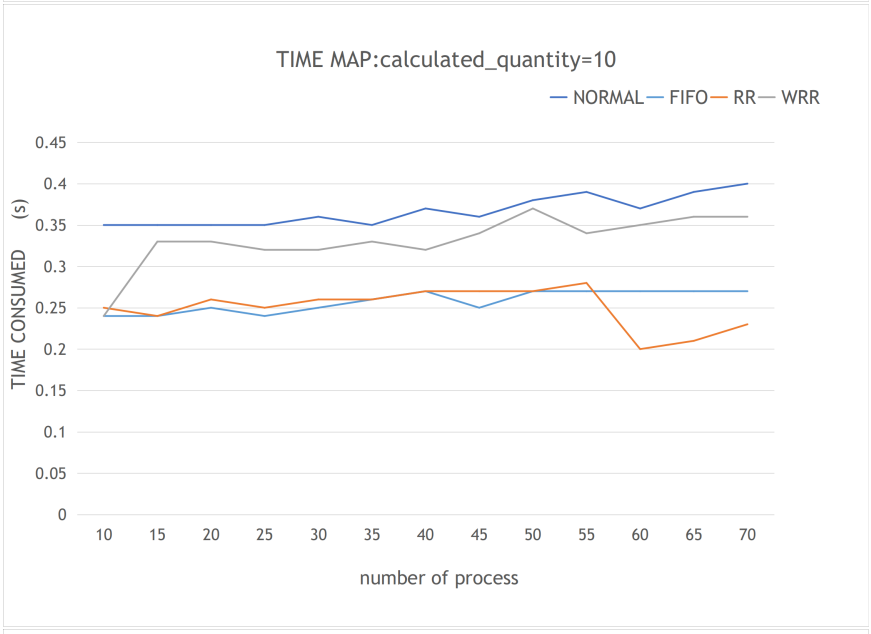
3.2 result and Analysis

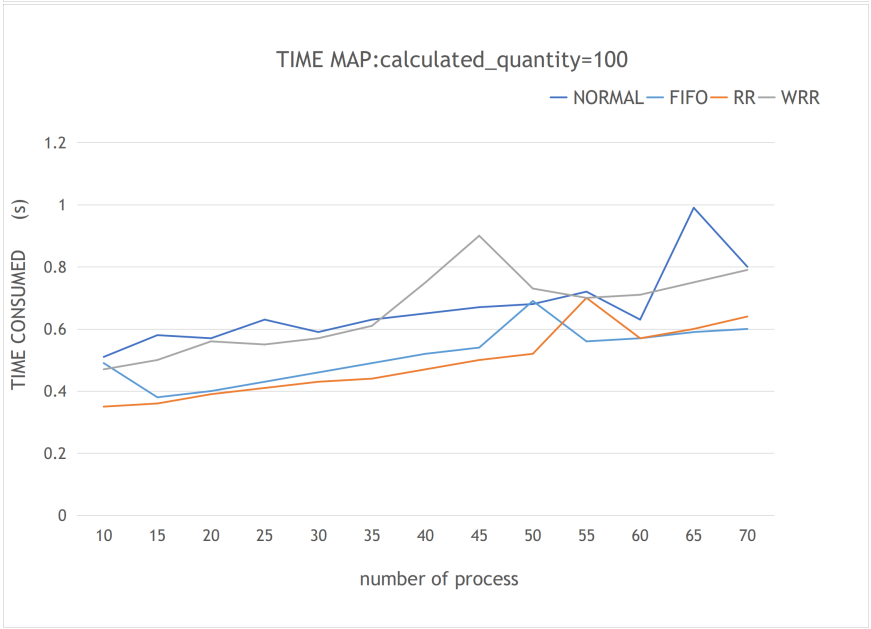
Set the ”calculated_quantity” at 1、10、100、1000、2000、3000 and ”num_childproc” from 10 to 70. We got 6 chart showing the time cost of each schedule policy.

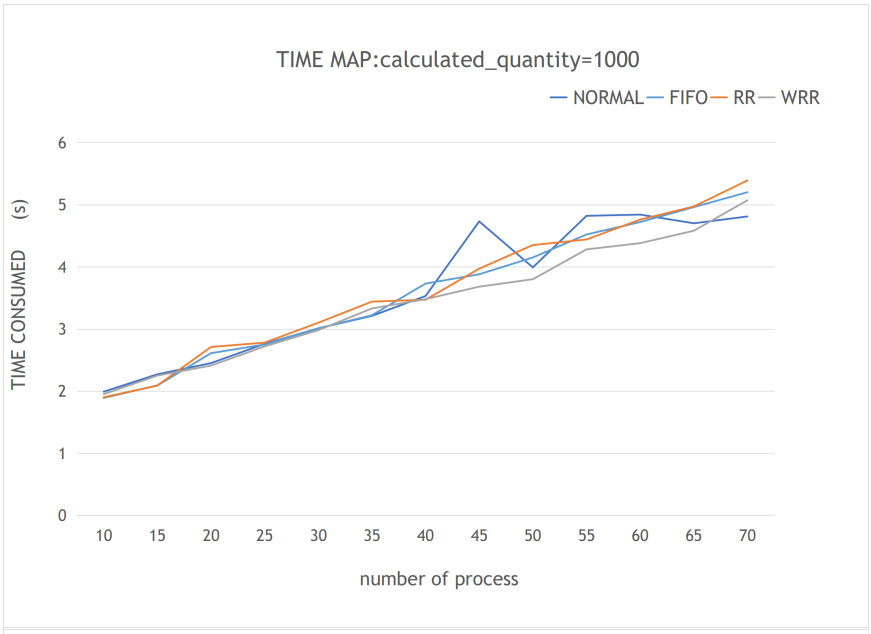
The performance of WRR will be affected by the number of task in queue and the time needed of each task. We allocate 100ms as a time slice to the foreground task, if the task can not finish it work within 100ms then it will be preempted.

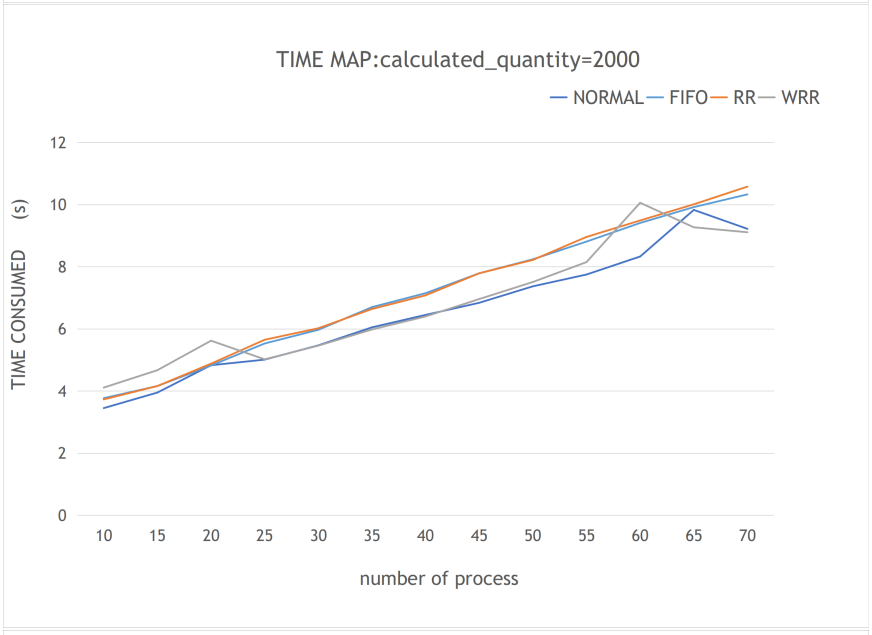
As we can see from the charts, WRR’s performance is close to NORMAL, also is worse than FIFO and RR when the calculated_quantity is low. I think thats because WRR takes more time to check the task group and set time slice, but actually it only needs very short time to finish the task.When task become harder and more time is needed, WRR is obviously faster than FIFO and RR. The reason for this result, i think could be that WRR gives the foreground task far more time than background task, so it looks like the task get a super high priority in the core, and that lead to it’s better performance.

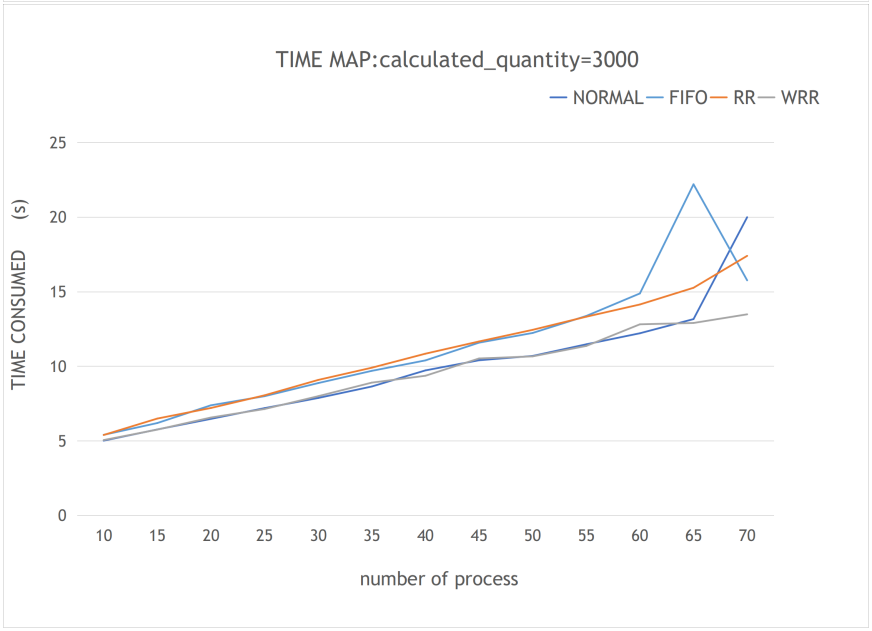

TIME MAP:calculated_quantity=1
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
10 15 20 25 30 35 40 45 50 55 60 65 70
number of process


TIME MAP:calculated_quantity=10
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
10 15 20 25 30 35 40 45 50 55 60 65 70
number of process


TIME MAP:calculated_quantity=100
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
1.2
1
0.8
0.6
0.4
0.2
0
10 15 20 25 30 35 40 45 50 55 60 65 70
number of process


TIME MAP:calculated_quantity=1000
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
0
1
2
3
4
5
6
10
15
20
25
30
35
40
45
50
55
60
65
70
number of process


TIME MAP:calculated_quantity=2000
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
0
2
4
6
8
10
12
10
15
20
25
30
35
40
45
50
55
60
65
70
number of process


TIME MAP:calculated_quantity=3000
NORMAL
FIFO
RR
WRR
TIME CONSUMED (s)
0
5
10
15
20
25
10
15
20
25
30
35
40
45
50
55
60
65
70
number of process