OS Project 2 Report

Introduction

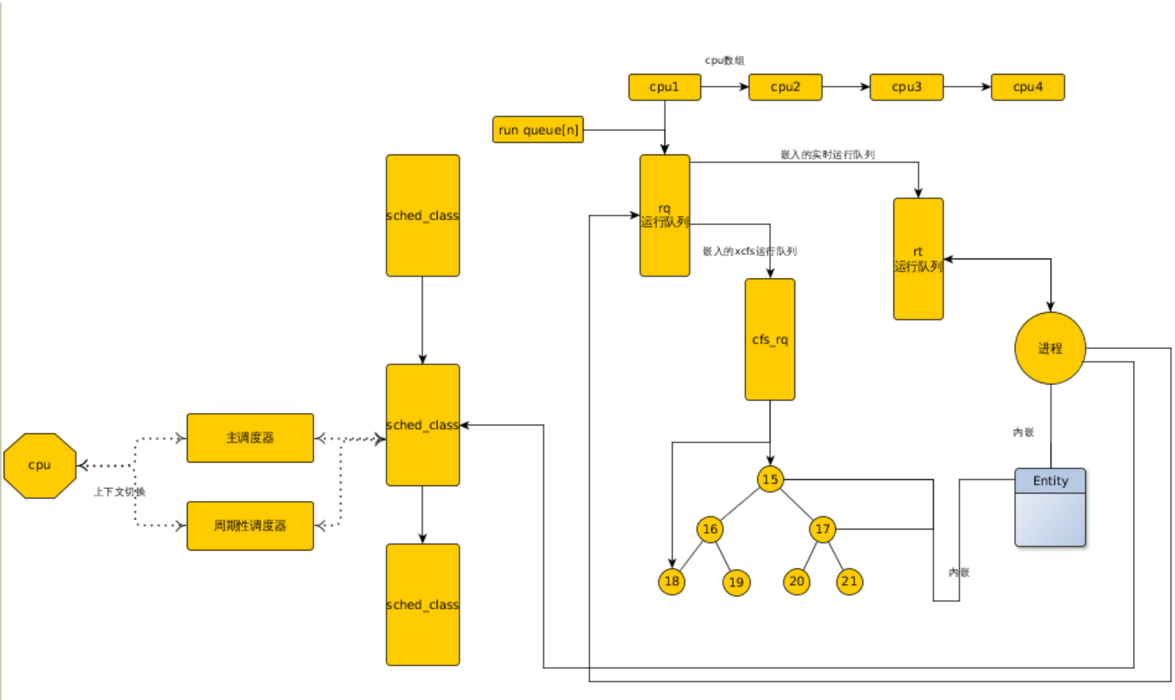
CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this project, we are asked to implement a particular CPU scheduler for Linux based Android operating system.

Our new Android scheduler is a case-distinct Round-robin schedule policy ---- weighted Round-robin (WRR for short). WRR will assign more execution time unit for foreground tasks and less time for background tasks.

Problem Analysis & Implementation

To better understand how linux scheduler works, especially how the existing RR scheduler works, we need to refer the provided website, and to see the source code in "/kernel/sched" like "core.c" and "rt.c".

After some investigation, we can get a big picture of the Linux scheduler structure, like the figure below.



So our job is to write a new sched_class similar with existing rt and cfs sched_classes. And define corresponding running queue struct and entity structure to store some necessary information.

Based on analysis above, we have the following implementation:

- Modify some existing files:

- goldfish_armv7_defconfig : add `CONFIG_WRR_GROUP_SCHED` option.
- /include/linux/sched.h :
  - define `SCHED_WRR` with const value 6 to indicate our WRR policy.
  - define `WRR_BG_TIMESLICE` & `WRR_FG_TIMESLICE` with const value 10ms and 100ms respectively.
  - define `sched_wrr_entity` structure to store some necessary information for each wrr task. The information contains:
    - `timeslices`: used as a counter to indicate how much time per RR turn.
    - `weight` : to indicate the state of a task: when a task is in foreground, its `weight` will be 10, when it is in background, its `weight` will be 1. This weight is also used for load-balance purpose.
    - `run_list` : a `link_head` instance to link neighbor tasks on the same running queue.

  In addition to these definitions, we also need to declare some wrr variables in proper places:

  - add `sched_wrr_entity` member `wrr` to `task_structure`.
  - declare `wrr_rq` structure.
- /kernel/sched/sched.h :
  - define `wrr_rq` structure, and add it to `rq` structure. As running queue, `wrr_rq` maintains general information of `wrr_entity`s, like `wrr_nr_running` (the number of tasks in this queue), `total_weight`.

    Unlike `rt_rq` , we don't define `prio_array`. `prio_array` need to work with `bitmap`. If we include `prio_array` struct in WRR, our project will be more difficult for coding and rebalancing.
  - declare some external function which will be implemented later on.
- /kernel/sched/core.c :
  - revise function `__sched_fork()`, `sched_fork()`, `wake_up_new_task()`, `scheduler_tick()`, `rt_mutex_setprio()`, `__setscheduler()`, `__sched_setscheduler()`, `sched_init()` to make WRR has the same behavior with RT and FAIR.
- /kernel/sched/rt.c
  - modify the struct vaiable `.next` in `rt_sched_class` to make it point to `wrr_sched_class`
- /include/linux/init_task.h :
  - add `.wrr` struct for initialization.
- /kernel/sched/Makefile ：
  - add `wrr.o` for compilation.

- Create /kernel/sched/wrr.c. It contains `wrr_sched_class`, which is major part of WRR. The main idea of implementing this class and related functions is to imitate and revise codes in rt.c. We omit all codes related to `SMP` and `Preemption` and substitute `bitmap` and `prio_array` struct operations with in-built `list` operations like `list_add` and `list_del`.

  The things we need further add to `wrr_sched_class` is the codes related to judging the state of tasks, and switching the timeslice the tasks' states changing. To implement this we have to use function `task_group_path()` in /sched/debug.c to get the state of a task. We add the state checking code in function `enqueue_wrr_entity()`.

Description above is the general idea of the implementation. After these modifications and creations, we can type `make -jx`(x indicates number of core used) to get the target image file `zImage`.

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.

Before testing, we need write a test program to set a task's schedule policy. So we write a program `set_sched` getting some necessary input for changing schedule policy. This program uses in-build syscall `sched_setscheduler()` to finish switching.

Besides, we also write a program `wrr_info` by using syscall `sched_getscheduler()` and `sched_rr_get_interval()` to a task's schedule policy information and execution time interval.

We randomly select an APP in Android emulator and run it. By typing `ps -P | grep [APP's name]`, we can get sufficient information for `set_sched` . After running `set_sched`, we can get the information from kernel message like fig below. Apparently, the target task has switched to WRR schedule policy.

```
I AM IN __sched_setscheduler.
group = /
Finish switch_to_wrr pid: 1252 !
Switched to a foreground wrr pid: 1252, proc: est.processtest
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 10
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 9
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 8
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 7
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 6
```

If we further run `wrr_info`, we will get the following information. The task is in foreground with timeslice 100ms.

```
$ adb shell
root@generic:/ # cd data/misc
root@generic:/data/misc # ps -P | grep processtest
u0_a53    1252  84    502788 43820 fg   sys_epoll_ aa0f8478 S com.osprj.test.processtest
root@generic:/data/misc # ./set_sched
Input the process id (PID) you want to modify: 1252
Input the schedule policy you want to change (0-NORMAL, 1-FIFO, 2-RR, 6-WRR):
6
Set process's priority: 0
Changing Scheduler for PID 1252
Switch finish!
10|root@generic:/data/misc # ./wrr_timeslice
Input the process id (PID) you want to check: 1252
Schedule policy: wrr
Timeslice: 100 milisec                        Click home button
23|root@generic:/data/misc # ./wrr_timeslice
Input the process id (PID) you want to check: 1252
Schedule policy: wrr
Timeslice: 10 milisec
```

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.

```
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 3
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /
  cpu: 0 task_tick: 1252 time_slice: 2
Change timeslice to background
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /bg_non_interactive
  cpu: 0 task_tick: 1252 time_slice: 1
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /bg_non_interactive
  cpu: 0 task_tick: 1252 time_slice: 1
! Enter task tick 6, pid: 1252, number of wrr proc: 1
task_tick_wrr: task_group: /bg_non_interactive
  cpu: 0 task_tick: 1252 time_slice: 1
```

Obstacles

To implement wrr scheduler, we really need read a lot of codes and refer many documents to understand its mechanism. This project is very different from projects we have done before. We are asked to create something new, but we also need to follow kind of strict programming rule to make WRR work in the whole operating system.

Firstly, we need to figure out reliance of different code files, which require us to read many related materials. Luckily, I find some useful tips from [stack overflow](), which shows almost every related files.

Secondly, problems come from compiling stage. For the first several times compilations, I encountered many errors, such as inproper substitution of "rt" variables, lack of declaration, implicit reliance.

Then, problems arise from switching stage. When I lauched an APP and prepared to run `set_sched` program, the APP crashed with no respond to my `setscheduler` function. This took me a lot of time to solve. Finally, I found the problem was in rt.c file. 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.

After that I still got problem in `setscheduler`, this time I received kernel-panic message. I really didn't know how to deal with it. This incorrect `wrr_sched_class` uses similar structure with `rt_sched_class` 's, so it contains `prio_array`, `bitmap` and `rt_bandwidth`. By trials and errors, I decide to omit these functions and try to make my `wrr_sched_class` be simple and understandable.

By now my own `wrr_sched_class` finally works.

Additional Work/Bonus

My additional work mainly focuses on multi-cpu archecture, actually symmetric multi-processor archecture (SMP for short), and load balance among multiple cpus.

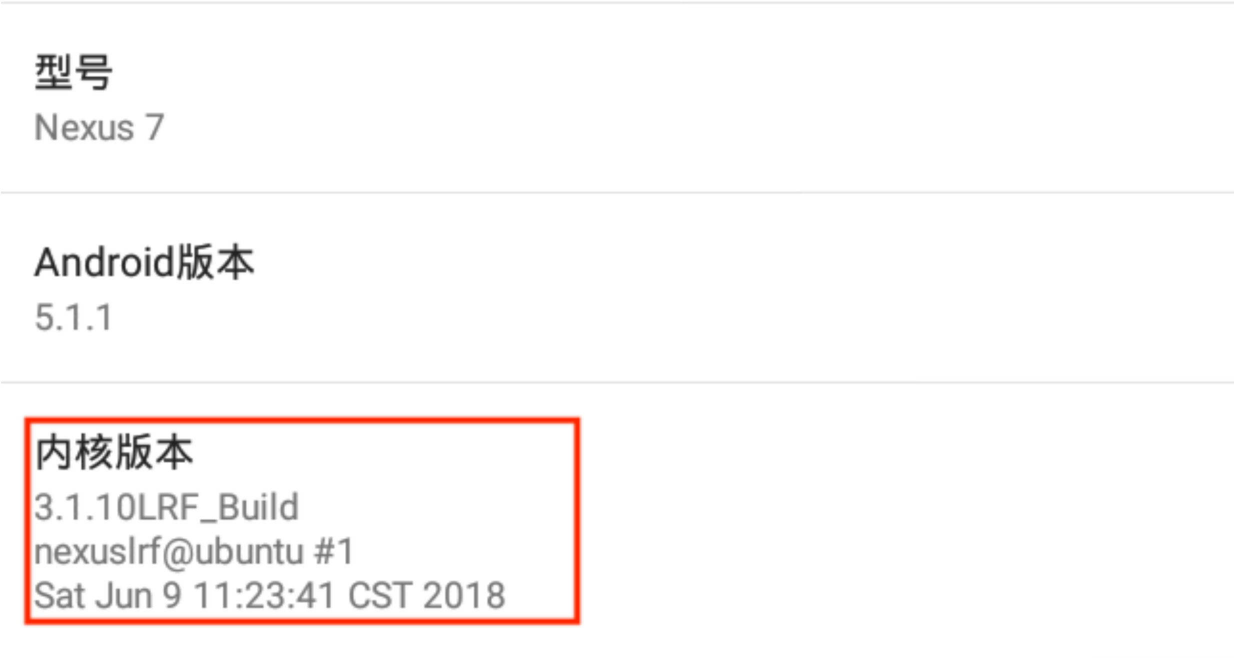
Due to our provided emulator is a uniprocessor archecture, which is meaningless for this work. And now it turn to use some real devices for development. Luckily, I have a [Google Nexus 7 (2012)](#) tablet, which is bought nearly 5 years ago. This tablet use an ARMv7 quad-core CPU which is embedded in the Nvidia Tegra 3 SoC, so it can well support SMP functions.

Load a kernel on tablet will be another huge work. The kernel our emulator uses cannot load to a real devices for lacking some necessary lower layer API. So we need download a new kerrnel, and do all the work over again. The kernel file can be got from [AOSP web](#).

The tablet uses a linux kernel with 3.1.10 version, which is earlier than our emulator's goldfish version (3.4.67), so some files have different name and some struct have different definitions. Therefore, I need to write a new WRR scheduler from scratch.

After writing the normal WRR scheduler, I take a easy method to support SMP load balance — when a new wrr task comes to execution, `wrr_sched_class` select an available CPU with the smallest `wrr_rq.total_weight` value and assign new task to this CPU. This feature is implemented by `wrr_sched_class` member function `select_task_rq_wrr`.

After finishing these modification, I start to build my kernel and load it onto my tablet. This process follows the tutorial from [packtpub](#) and [stackexchange](#). Then the tablet will run my own kernel.



By doing some normal tests, I find that WRR scheduler works fine for this real device. Next, I test how it works for SMP case.

This time, I write a dummy program which runs a meaningless `while` loop. I run 8 dummy programs at the same time (If I run more, my device is too busy to deal with them).

```
root@grouper:/data/misc # ./dummyARM
Input number of dummy: 8
7538
chd pid 0: 7554
chd pid 1: 7555
chd pid 2: 7556
chd pid 3: 7557
chd pid 4: 7558
chd pid 5: 7559
chd pid 6: 7560
chd pid 7: 7561
```

```
Selecting task_rq:
cpu: 0 nr_running: 1 total_weight: 10
cpu: 1 nr_running: 1 total_weight: 10
cpu: 2 nr_running: 1 total_weight: 10
cpu: 3 nr_running: 1 total_weight: 10
get_wrr_interval_Task_group: /
task_tick_wrr: task_group: /
task_tick_wrr: task_group: /
task_tick_wrr: task_group: /
task_tick_wrr: task_group: /
 cpu: 3 pid: 7559 proc: dummyARM time_slice: 7
 cpu: 0 pid: 7561 proc: dummyARM time_slice: 8
 cpu: 1 pid: 7555 proc: dummyARM time_slice: 3
 cpu: 2 pid: 7557 proc: dummyARM time_slice: 2
```

As you can see these 4 dummy programs run in different CPUs at the same time, which shows my load balance policy works.

Achievements

From this project, I learnt a lot about the linux kernel programming, and experienced the combination of hardware device and software system. This gives me a great impression on the structure of Android operating system. I think this project is beneficial for my future programming development.