# Homework 4

## W4118 Fall 2014

**UPDATED: Monday 11/03/2014 at 12:12pm EST**
**DUE: Thursday 11/06/2014 at 11:59pm EST**

All non-programming, written problems in this assignment are to be done by yourself. Group collaboration is permitted only on the kernel programming problems. All homework submissions (both individual and group) are to be made via **Git**.

You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Be aware that commits pushed after the deadline will not be considered. Refer to the homework policy section on the **class web site** for further details.

## Individual Written Problems:

The Git repository you will use for the individual, written portion of this assignment can be cloned using:

```
git clone https://os1.cs.columbia.edu/UNI/hmwk4-writ.git
```

(replace `UNI` with your own UNI). This repository will be accessible only by you, and you will use the same SSH public/private key-pair used for Homework 1.

Exercise numbers refer to the course textbook, *Operating Systems Principles and Practice*. Each problem is worth 5 points.

1. Exercise 7.4 (Chapter 7, Exercise 4)

2. Exercise 7.13 (Use the example in the book in Figure 7.5 and assume that the time slice for each lower priority level is double the previous one and that a task that yields remains at the same priority level.)

3. Exercise 7.16

4. Exercise 7.17

5. When a process forks a child, both parent and child processes are runnable. Assuming there are no other processes in the system and a single CPU system, explain how the Linux 3.4 default scheduler will schedule the parent and child processes, including which process will run after the execution of fork.

6. Explain how load balancing is done in the realtime scheduler in Linux 3.4.

## Group Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository your entire group will use to submit the group programming problems can be cloned using:

```
git clone https://os1.cs.columbia.edu/TEAM/hmwk4-prog.git
```

(Replace `TEAM` with the name of your team, e.g. `team1`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as `git-pull`, `git-merge`, `git-fetch`. You can see the documentation for these by writing `$ man git-pull` etc. You may need to install the `git-doc` package first (e.g. `$ apt-get install git-doc`.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the scripts/checkpatch.pl script included in the Linux kernel. Errors from the script in your submission will cause a deduction of points.

All kernel programming assignments in this year's class are done on the Android operating system and targeting the ARM architecture. For more information on how to use adb and aliasing refer to Homework 2. Use "adb -e" for emulator commands and "adb -d" to direct the commands to the device. If you run in to a read-only disk error then type "adb remount" add -e or -d accordingly.

You can use Homework 2's VM for this assignment, which can be downloaded from here [ www.cs.columbia.edu/~nieh/teaching/w4118/homeworks/w4118.f14.v4.linux.zip ] .

The kernel programming for this assignment will be done using a Google **Nexus 7 (2013)** The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Google Nexus 7 uses an ARMv7 CPU. The Android emulator can also be used and is built on a system called QEMU which emulates an ARMv7 processor.

## The goal of this assignment is to make a new linux scheduler, a Grouped Round-Robin. As a first step, you are to implement a simple round robin scheduler in part 1, and add group features in part 2.

1. **(30 pts.) A Multicore Round-Robin Scheduler**
   Add a new scheduling policy to the Linux kernel to support *round-robin* scheduling. Call this policy GRR. The algorithm should run in constant time and work as follows:
   i. The new scheduling policy should serve as the default scheduling policy for `init` and all of its descendants.
   ii. Multicore systems, like the Nexus 7, must be fully supported.
   iii. Every task has the same time slice (quantum), which is 100ms.
   iv. When deciding which CPU a task should be assigned to, it should be assigned to the CPU with the smallest total number of tasks in the CPU's run queue.
   v. Periodic load balancing should be implemented such that a single job from the run queue with the highest total number of tasks should be moved to the run queue with the lowest total number of tasks. The job that should be moved is the first eligible job in the run queue which can be moved without causing the imbalance to reverse. Jobs that are currently running are not eligible to be moved and some jobs may have restrictions on which CPU they can be run on. Load balancing should be attempted every 500ms for each CPU; you may assume for load balancing that the CPUs receive periodic timer interrupts.
   - The Linux scheduler implements individual scheduling classes corresponding to different scheduling policies. For this assignment, you need to create a new scheduling class, `sched_grr_class`, for the `GRR` policy, and implement the necessary functions in

`kernel/sched/grr.c`. You can find some good examples of how to create a scheduling class in `kernel/sched/rt.c` and `kernel/sched/fair.c`. Other interesting files that will help you understand how the Linux scheduler works are `kernel/sched/core.c` and `include/linux/sched.h`. While there is a fair amount of code in these files, a key goal of this assignment is for you to understand how to abstract the scheduler code so that you learn in detail the parts of the scheduler that are crucial for this assignment and ignore the parts that are not.

- Your scheduler should operate alongside the existing Linux scheduler. Therefore, you should add a new scheduling policy, `SCHED_GRR`. The value of `SCHED_GRR` should be 6. `SCHED_GRR` should be made the default scheduler class of `init`.
- Only tasks whose policy is set to `SCHED_GRR` should be considered for selection by your new scheduler.
- Tasks using the `SCHED_GRR` policy should take priority over tasks using the `SCHED_NORMAL` policy, but *not* over tasks using the `SCHED_RR` or `SCHED_FIFO` policies.
- Your scheduler must be capable of working on both uniprocessor systems (like the emulator) and multicore/multiprocessor systems (like the Nexus 7). All cores should be utilized on multiprocessor systems.
- Proper synchronization and locking is crucial for a multicore scheduler, but not easy. Pay close attention to the kind of locking used in existing kernel schedulers.
- For a more responsive system, you may want to set the scheduler of kernel threads to be `SCHED_GRR` as well (otherwise, `SCHED_GRR` tasks can starve the `SCHED_NORMAL` tasks to a degree). To do this, you can modify kernel/kthread.c and replace `SCHED_NORMAL` with `SCHED_GRR`. You don't have to though, this is not a requirement.

2. **(30 pts.) An Android group scheduler**

Round-robin scheduling treats all tasks equally, but there are times when it is desirable to give some tasks preference over others. In Android, tasks are classified into different groups so that different task groups can be treated different. Modify your scheduler from Part 1 to add group scheduling features that can be used by Android.

Android 4.4 has three groups of tasks: system group, foreground group, background group. Typically, Android tasks are assigned to the foreground or background group, and kernel threads are in the system group. From user level, you can run `ps -P` on the device or emulator to check the assigned groups for each task. To get information for a specific pid, you can run `cat /proc/{pid}/cgroup`. Observe the change of a task's group when you launch an app and go back to the launcher by pressing the home button. The `add_tid_to_cgroup` function in **this file** will help you to understand how Android set tasks to each group. At the kernel level, a task's group information can be found using a task pointer. Refer to the line 136 in `kernel/sched/debug.c` and use it appropriately. The return value will be "/" for a system group, "/apps" for a foreground group, "/apps/bg_non_interactive" for a background group. Whenever Android changes the group of a task, `cpu_cgroup_attach` function in `kernel/sched/core.c` will be called.

In this assignment, you will introduce group scheduling using two groups, a foreground group and a background group, and adjust the number of CPU cores assigned to each group to change their performance characteristics:

- Two groups exist in this scheduler: a foregound and a background group. Your scheduler should treat Android's system and foreground groups together as the foreground group, and Android's background group as the background group.
- Tasks either belong to foreground or background group, and each group of tasks run on

dedicated set of cores. Android framework sets tasks as foreground or background according to the status of an app. Some more explanation is found below.

- At the beginning by default, half of cores are configured to run foreground tasks and the other half of cores are configured to run background tasks. In the case of uniprocessor systems, a core can run both of foreground tasks and background tasks. Assigning some number of cores to specific group will be made possible at runtime via a system call `sched_set_CPUgroup` which you are to implement. Note that for SMP/multicore, at least one CPU should be always be assigned to each group.
- When a task becomes runnable, it should be assigned to the core that is least busy, which is the one with the shortest runqueue.
- If a core has nothing to do, try to steal a task from another core in the same group. If there is nothing to steal, let the core remain idle.
- Periodic load balancing is done within cores dedicated to the same group with the current core.

**Assigning CPUs to group**:
For assigning CPUs to a group, you are to implement the following system call:

```
#define FOREGROUND 1
#define BACKGROUND 2

/* This system call will assign numCPU to the given group,
 * and assign (totalCPU - numCPU) to the other group.
 * System call number 378.
 */
int sched_set_CPUgroup(int numCPU, int group);
```

Only the administrator (`root` user) may adjust CPU configuration using `sched_setCPUgroup()`. The system call should handle all errors appropriately. (e.g. assigning 0 CPU to any of the groups) The system call should be implemented in `kernel/sched/core.c`.

3. **(10 pts.) Investigate and Demo**
   Demonstrate that your scheduler works with Android apps and show how changing the number of CPUs assigned to the foreground and background groups affects app performance. For example, for a given workload, what happens if you assign 3 CPUs to the foreground group versus assigning 1 CPU to the foreground group? If you assign 3 CPUs vs 1 CPU to the foreground group, does the foreground app run 3 times as fast? Why or why not? What qualitative difference in performance do you see when running a workload with different CPU configurations? Using a benchmark app is a good idea. You should download apps from Google Play as needed and try running different mixes of those apps. Geekbench and Quadrant are popular benchmarks, however feel free to use any apps which may be useful to show any performance difference. Many Android apps normally stop running when they are in the background, so you will need to select your apps carefully to demonstrate the impact of scheduling. Some apps you might consider for running in the background include the Pi benchmark and some anti-virus apps. How does your scheduler compare to the default Android scheduler in terms of qualitative performance? Are there any scenarios in which there are visible performance differences?

   You should provide a complete set of results that describe your experiments and be prepared to demonstrate those scheduling scenarios to the TAs. Your results and any explanations should be put in the `written` file in the root of your team's `hmwk4` repository. The writeup should be of sufficient detail for someone else to be able to repeat your experiments.

## Additional Hints/Tips

### Kernel / Scheduler Hacking:

- You may want to refrain from immediately making your scheduler the default scheduler for `init` and instead, test by manually configuring tasks to use your policy with `sched_setscheduler()`.
- When debugging kernel crashes on the device, once the device reboots after a crash, the kernel log from that crash will exist in `/proc/last_kmsg` (provided it was a soft reboot). Consider booting the device with your custom kernel using `fastboot boot` instead of flashing the new kernel. Then if it crashes, it will reboot into a stable kernel and you can view `last_kmsg`.
- For this homework, the default kernel configurations for both the emulator and the device have been updated to include `debugfs`, and some basic scheduler debugging. These tools can be of great value as you work on this assignment. Debugfs documentation can be found **here**, and scheduler debugging information can be found in `/proc/sched_debug` and `/proc/schedstat`. You can also search through the code for `SCHED_DEBUG` and `SCHEDSTATS` - you may want to add something while you debug! debugfs can be mounted with `mount -t debugfs none /sys/kernel/debug` if not already mounted.
- Please apply the **kernel patch** as discussed on **Piazza**.