**Homework 4**

**W4118 Fall 2012**

**DUE: Wednesday, 11/7/2012 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](http://git-scm.com/). Git repository access will use the same public/private key-pair you used for previous homeworks (see these [Git](http://www.cs.columbia.edu/%7Enieh/teaching/w4118/homeworks/git_guide.html) instructions).

**Individual Written Problems:**

The Git repository you will use for the individual, written portion of this assignment can cloned using: git clone gitosis@os1.cs.columbia.edu:UNI/hmwk4.git (replace UNI with your own UNI). This repository will be accessible only by you.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points.

1. Exercise 5.12 but assuming a 2-CPU system scheduled from a single runqueue.
2. Exercise 5.16
3. Exercise 5.17
4. Exercise 5.21
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 3.1 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 3.1.

**Group Programming Problems:**

Group programming problems are to be done in your assigned [groups](http://www.cs.columbia.edu/%7Enieh/teaching/w4118/homeworks/groups.pdf). The Git repository your entire group will use to submit the group programming problems can be cloned using: git clone gitosis@os1.cs.columbia.edu:TEAM/hmwk4.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](http://www.kernel.org/pub/software/scm/git/docs/git-pull.html), [git-merge](http://www.kernel.org/pub/software/scm/git/docs/git-merge.html), [git-fetch](http://www.kernel.org/pub/software/scm/git/docs/git-fetch.html). One important thing to remember is to pull any changes into your local repository before trying to push any commits to the class server.

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. Please follow the [Linux kernel coding style](http://www.kernel.org/doc/Documentation/CodingStyle).  
**Hint(1)**: use the checkpatch.pl script to ensure your code conforms to this standard. *Points will be deducted for non-compliance!*

It's a good idea to clone your repository after your final push (but before the deadline) and test the cloned code to be sure it behaves as expected.

All kernel programming assignments in this year's class are done on the Android operating and targeting the ARM architecture. For more information on how to get your development environment configured, please see this page: [Development Guide.](http://www.cs.columbia.edu/%7Enieh/teaching/w4118/homeworks/dev_guide.html)

The kernel programming for this assignment will be done using a [Nexus 7](http://en.wikipedia.org/wiki/Nexus_7) tablet. The Android platform can run on many different architectures, but the specific platform we will be targeting is the ARM CPU family. The Nexus 7 tablets use an ARMv7 quad-core CPU which is embedded in the Nvidia Tegra 3 SoC.   
  
Because the target CPU is different from the CPU running in your personal computer, you will have to *cross-compile* any software, including the linux kernel, to run on the different platform. You should use the same [virtual machine](http://os1.cs.columbia.edu/files/w4118.f12.linux.console.tar.bz2) you used for homeworks 2 and 3.

1. **(60 pts.) A Symmetric Multiprocessor Weighted Round-Robin Scheduler**

Add a new scheduling policy to the Linux kernel to support *weighted round-robin* scheduling. Call this policy WRR. The algorithm should run in constant time and work as follows:

* + 1. The new scheduling policy should serve as the default scheduling policy for init and all of its descendants.
    2. Multiprocessor systems, like the Nexus 7, must be fully supported.
    3. The base time slice (quantum) should be 10ms. Weights of tasks can range between 1 and 20 (inclusively). A task's time slice is determined by its weight multiplied by the base time slice. The default weight of tasks should be 10 (a 100ms time slice).
    4. If the weight of a task currently on a CPU is changed, it should finish its time quantum as it was before the weight change. i.e. increasing the weight of a task currently on a CPU does not extend its current time quantum.
    5. When deciding which CPU a job should be assigned to, it should be assigned to the CPU with the smallest total weight (i.e. sum of the weights of the jobs on the CPU's run queue).
    6. Periodic load balancing should be implemented such that a single job from the run queue with the highest total weight should be moved to the run queue with the lowest total weight, provided there exists a job in the highest run queue that can be moved to the lowest run queue without causing the lowest run queue's total weight to become greater than or equal to the highest run queue's total weight. The job that should be moved is the highest weighted eligible job which can be moved without causing the weight 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.
  + The Linux scheduler implements individual scheduling classes corresponding to different scheduling policies. For this assignment, you need to create a new scheduling class, sched\_wrr\_class, for the WRR policy, and implement the necessary functions in kernel/sched\_wrr.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.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\_WRR. The value of SCHED\_WRR should be 6. SCHED\_WRR should be made the default scheduler class of init.
  + Only tasks whose policy is set to SCHED\_WRR should be considered for selection by your new scheduler.
  + Tasks using the SCHED\_WRR policy should take priority over tasks using the SCHED\_NORMAL policy, but *not* over tasks using the SCHED\_RR or SCHED\_FIFO policies.
  + The weight of a task and the SCHED\_WRR scheduling flag should be *inherited* by the child of any forking task.
  + Your scheduler must be capable of working on both uniprocessor systems (like the emulator) and multiprocessor systems (like the Nexus 7). All cores should be utilized on multiprocessor systems.
  + Proper synchronization and locking is crucial for an SMP scheduler, but not easy. Pay close attention to the kind of locking used in existing kernel schedulers.

**Setting / Getting Weights**:  
For setting and getting the weights, you are to implement the following system calls:

/\* Set the SCHED\_WRR weight of process, as identified by 'pid'.

\* If 'pid' is 0, set the weight for the calling process.

\* System call number 376.

\*/

int sched\_setweight(int pid);

/\* Obtain the SCHED\_WRR weight of a process as identified by 'pid'.

\* If 'pid' is 0, return the weight of the calling process.

\* System call number 377.

int sched\_getweight(int pid);

The system calls should handle errors appropriately.

1. **(10 pts.) Investigate**  
   Demonstrate that your scheduler works with a test program that calculates the prime factorization of a number using the naive *Trail Division* method. Track how long this program takes to execute with different weightings set and plot the result. You should choose a number to factor that will take sufficiently long to calculate the prime factorization of, such that it demonstrates the effect weighting has on its execution time. Timing of the execution can be done either internally in the program's code or externally so long as it is sufficiently accurate to show the effect of weighting.  
     
   You should provide a complete set of results that show all your tests. If there are any results that do not yield execution time proportional to weights, explain why. Your results and any explanations should be put in the written file in the root of your team's hmwk4 repository. Your plot should be named plot.pdf and should be put in the root of your team's hmwk4 repository (next to the written file).

**Additional Hints/Tips**

**Kernel / Scheduler Hacking:**

* Remember that you must configure your kernel before building, and whenever you switch between the emulator and the Nexus 7 device. The command to configure the kernel for the device is: make ARCH=arm CROSS\_COMPILE=arm-none-linux-gnueabi- tegra3\_android\_defconfig and similarly for the emulator: make ARCH=arm CROSS\_COMPILE=arm-none-linux-gnueabi- goldfish\_armv7\_defconfig
* For this homework the default kernel configurations for both the emulator and the device have been updated to include the 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: [here](http://os1a.cs.columbia.edu/lxr/source/Documentation/filesystems/debugfs.txt), and scheduler debugging information can be found in /proc/sched\_debug and /proc/schedstat respectively. You can also search through the code for SCHED\_DEBUG and SCHEDSTATS - you may want to add something while you debug! The debugfs can be mounted with mount -t debugfs none /sys/kernel/debug if not already mounted.
* 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.

**Linux users:** If you are a Linux user and cannot connect your device to your VM to flash the system (or you would just like a more efficient set up), you may want to develop on your Linux host machine. While we do not have the resources to support your desktop installations, you may find this more convenient. To make it work, you should follow these steps:

1. Obtain the Android SDK from your VM. It's located at /home/w4118/utils/android-sdk\_r20.0.3-linux-jellybean.tgz
2. Extract it to ~/ (once extracted, there will be a ~/android-sdk-linux directory)
3. Add ~/android-sdk-linux/tools and ~/android-sdk-linux/platform-tools to your path (e.g. in ~/.bashrc)
4. Download the code-sourcery cross-compiler from [here](https://sourcery.mentor.com/sgpp/lite/arm/portal/package7851/public/arm-none-linux-gnueabi/arm-2010.09-50-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2)
5. Extract this tar somewhere (e.g. ~/tools)
6. Add the extracted arm-2010.09/bin directory to your path (e.g. in ~/.bashrc)
7. Set the CROSS\_COMPILE variable to arm-none-linux-gnueabi-

Hint: You can use your VM as a guide by looking at its ~/utils directory and ~/.bashrc file.