Lab3 – Implementing the Fair-Share Scheduling Policy

Due: 11:59pm Monday, Oct. 08

Instruction: In this lab, you will implement the fair-share scheduling. Fair-share scheduling is to equally distribute CPU time among users or groups in the system, as opposed to equal distribution among processes. For example, if user A and B launch 2 and 3 processes respectively, user A and B will have 50% of the entire CPU time. Each of A's processes will have 25% of the CPU time since A has 2 processes. For B, each process will have 50% / 3 = 16.67% of the CPU time. This policy dynamically enforces fair shares among different users. The CPU time will be re-distributed when a new process arrives or an old process terminates. For example, if one of A's processes terminates, the other one will have 50% of the CPU time.

You need to add a new scheduler policy to support fair-share scheduling. For all the normal processes (except the ones owned by *root* for performance reason) in the system, your new scheduler should apply fair-share scheduling policy based on the processes' user identities (you can check the *uid* in its task descriptor). To simplify the case, let us assume no real-time processes, i.e., no processes under the scheduling policy SCHED_RR or SCHED_FIFO (you can disable or nullify the system call *sched_setscheduler(…)*). To make your new scheduler policy easier to use, you need to add a system call that enables and disables the fair-share scheduling.

After implementing the scheduler, you need to write kernel instrumentation code for profiling the kernel performance in terms of the scheduler's behavior. For example, when the scheduler function is being executed, you need to dump relevant statistics such as the time when the scheduler is called, the process being scheduled out, the process being scheduled in, and the corresponding timeslices for both processes. Furthermore, you need to add a new system call that can enable or disable this scheduler profiling function. **Note that you CANNOT use the existing process information in /proc filesystem.**

Now, you should write a user-level program to post-process the profiled data, which is stored in a file. Based on the profiling data, draw a figure or table to show the way how the CPU time is distributed among existing processes and users within a configurable period of time. You can write this part of the code in either C or some script languages.

Test: To test your solution, you need create four new users A, B, C, D and write a CPU-bounded program, which can be an infinite loop. Then log into the system via these four users and run 3-7 instances of the test program for each user. Now you can enable the kernel profiling function by the new system call, wait for 1 minute, and disable the profiling function. Lastly, draw the figure or table using your post-processing program. If the solution is correct, you will find that the CPU time for A, B, C, D should be equally distributed.

Submission: First, you need to write a report about your solution. In this report, you should explain your design, implementation, and discuss the changes you made to the kernel and reasons for the changes.

Second, you need to make a diff file between your new kernel and the original kernel source code (extracted from the compressed file in /usr/src/kernels). Then submit both the diff file and the report on Carmen.

You can use the following command to make a diff file.

```
$> diff -Naur linux-2.6.9 linux-2.6.9lab# > lab#.diff
```

Before making the diff file, you should clean up the source tree by issuing the command “make mrproper” (You may want to read the file “linux-2.6.9lab#/Makefile” to get more information about several clean-up options).

Cleanup: Since we do not have enough disk space in each virtual machine, you need clean up the virtual machine after my evaluation for each of your lab solutions. First, delete the lab working directory at /usr/src/kernels/linux-2.6.9lab#. Then, delete the directory of installed modules at /lib/modules/linux-2.6.9lab#.