Homework 4

W4118 Spring 2021

UPDATED: Saturday 3/6/2021 at 8:42pm EST
DUE: Thursday 3/18/2021 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. Please edit and include this **file** in the top-level directory of your homework submission. Homeworks submitted without this file will not be graded. 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:

Please follow the Github Classroom link: **Homework 4**. The will result in a GitHub repository you will use that can be cloned using

```
git clone git@github.com:W4118/s21-hmwk4-written-UserName.git
```

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

Please note that the link and the created repo is for the individual written part only. The workflow of this written part is the same as Homework 1. For group assignment, the workflow is different. Refer to the description below.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points. For problems referring to the Linux kernel, please use the same kernel version as you use for the group kernel programming assignment. Please include all your answers in a file named written.txt that is uploaded to your GitHub repository for this assignment. Graphs or charts can be uploaded as a separate PDF file, for which a link has been included in your written.txt that can be clicked on to open the file.

1. Exercise 6.11
2. Exercise 6.16
3. Exercise 6.22
4. Exercise 6.25
5. Exercise 6.28

Group Programming Problems:

Group programming problems are to be done in your assigned groups. The Git repository for your group has been setup already on Github. You don't need the Github Classroom link for the group assignment. It can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk4-teamN.git
```

Please use --recursive
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 latest version of **checkpatch**. Errors or warnings from the script in your submission will cause a deduction of points.

The kernel programming for this assignment will be done using your x86 VMs. You can continue using the same VM that you used for Homework 2.

(75 pts.) The goal of this assignment is to make a new Linux scheduler, a Weight Round-Robin scheduler. As a first step, you are to implement a simple weighted round robin scheduler in part 1 and add additional features in parts 2 and 3.

1. **A Multicore Round-Robin Scheduler**
   Add a new scheduling policy to the Linux kernel to support *weight round-robin* scheduling. Call this policy WRR. 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 must be fully supported.

   iii. Every task has a time slice (quantum), which is a multiple of a base 10ms time slice.

   iv. The multiple should be the weight of the process which can be set using Linux scheduling system calls.

   v. When deciding which CPU a task should be assigned to, it should be assigned to the CPU with the least total weight. This means you have to keep track of the total weight of all the tasks running on each CPU. If there are multiple CPUs with the same weight, assign the task to the lowest number CPU among CPUs with the same weight.

   - 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/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_WRR`. The value of `SCHED_WRR` should be 7. `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.
- Your scheduler must be capable of working on both uniprocessor systems and multicore/multiprocessor systems; you can change the number of cores on your VM for testing. 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. This is extremely critical for the next part of the assignment. It is very easy to overlook a key synchronization aspect and cause a deadlock!
- For a more responsive system, you may want to set the scheduler of kernel threads to be SCHED_WRR as well (otherwise, SCHED_WRR 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_WRR. It is strongly suggested that you do this to ensure that your VM is responsive enough for the test cases.

2. **Getting WRR info**
   To make it easier to monitor the status of your scheduler, you will implement the following system calls:

```
#define MAX_CPUS 8 /* We will be testing only on the VMs */
struct wrr_info {

        int num_cpus;
        int nr_running[MAX_CPUS];
        int total_weight[MAX_CPUS];
}

/* This system call will fill the buffer with the required values
 * i.e. the total number of CPUs, the number of WRR processes on each
 * CPU and the total weight of WRR processes on each CPU.
 * Return value will also be the total number of CPUs.
 * System call number 436
 */
int get_wrr_info(struct *wrr_info);

/* This system call will change the weight for the calling process.
 * Only a root user should be able to increase the weight beyond
 * the default value of 10.  The system call should return an error
 * for any weight less than 1.
 * System call number 437.
 */
int set_wrr_weight(int weight);
```

The two system calls should be implemented in `kernel/sched/core.c`.

3. **Add load-balancing features to WRR**
   You should have a working scheduler by now. Make sure you test it before you move on to this part. So far your scheduler will run a task only on the CPU that it was originally assigned. Let's change that now! For this part you will be implementing two balancing features. The first will be a periodic load balancing where you will try to move a task from the heaviest CPU to the lightest one and the second is called idle balance which is when a CPU will attempt to steal a task from another CPU when it doesn't have any tasks to run i.e. when its runqueue is empty.

   Load balancing is a key feature of the default Linux scheduler (Completely Fair Scheduler). While CFS follows a rather complicated policy to balance tasks and works to move more than one task in a single go, your scheduler will have a simple policy in this regard.

- 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. 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. While there is a quite a bit of code in the CFS implementation, look in particular for the functions that determine whether a task is eligible to moved to another core. Don't expect to move a task every periodic load balancing cycle.
- Idle balance is attempted when a processor has an empty runqueue. Again, take a look at the CFS implementation to figure out where this is taking place and how it is called from `kernel/sched/core.c`. The CPU that is idle should attempt to pull one task from another CPU. You should choose the CPU to pull from that has at least two tasks on its runqueue and has the greatest total weight, although the measure of greatest total weight can be an estimate and can reflect measurements at somewhat different times on different CPUs. You should again ensure that the task you are stealing is eligible to be moved and allowed to run on the idle CPU.

4. **Investigate and Demo**
Demonstrate that your scheduler effectively balances weights across all CPUs. The total weight of the processes running on each CPU should be roughly the same. You should start with a CPU-bound test program such as a while loop then try running more complex programs that do I/O to show that the scheduler continues to operate correctly. A test program that calls the system call should be included in the test directory. Experiment with different weights and see what impact they have.

You should provide a complete set of results that describe your experiments. Your results and any explanations should be put in the `README` 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 will 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()`.**
- Before testing, take a snapshot of your VM if you have not done so already. That way, if your kernel crashes or is unresponsive because of scheduler malfunction, you can restore the state of the VM prior to the problem.