Introduction

The outcome of this project is to implement a new CPU scheduler in the Linux kernel. The objectives of this project is to learn:

1. How to add a pluggable CPU scheduler in Linux kernel.
2. How to apply different scheduling policies to individual processes.
3. Learn the internal working of a CPU scheduler.
4. Implement a weighted round robin scheduling policy.

Project submission

For each project, create a gzipped file containing the following items, and submit it to jrao@uccs.edu via email. Include “CS4500/5500_Spring2016_ProjectX” in the title of your email, where “X” is the project number.

1. A report that briefly describes how did you solve the problems and what you learned.
2. The Linux kernel source files that your modified or created.

Each team should specify a virtual machine (VM) that the instructor can login to check the project results. Name the VM as “CS4500/5500_LastNameoftheFirstMember_LastNameoftheSecondMember” and create a new user account called instructor in your VM. Place your code in the home directory of the instructor account (i.e., /home/instructor). Make sure the instructor has access to your code. In your submission email, include your password for the instructor account.

*Disclaimer: This assignment is adapted from projects developed by Dr. Jason Nieh at Columbia University and Dr. Kai Shen at Rochester University.
Assignment description

The Linux scheduler repeatedly switches between all the running tasks on the system, attempting to give a fair amount of CPU time to each task. In this project, you will implement a weighted round robin (WRR) task scheduler in the Linux kernel. The WRR scheduler assigns each process a time quantum proportional to the weight of the process. For example, if process A has weight 1 and process B has weight 2, they will receive 33% and 67% of the total CPU time, respectively.

You tasks

1. Implement the new WRR task scheduler. You do not need to implement the multiprocessor load-balancing at this time and assume that all the WRR processes run on a single core. Most of the code of interest for this assignment is in `kernel/sched.c`, `include/linux/sched.h`, and the new scheduler you implement, say `kernel/sched_wrr.c`. These are probably not the only files you will need to look at, but they’re a good start. 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. For example, since we will not implement the load-balancing, you do not need to read the code between `#ifdef CONFIG_SMP` and the corresponding `#endif`.

2. Implement two system calls `get_wrr_weight()` and `set_wrr_scheduler()`, with the following prototypes:
   ```c
   int get_wrr_weight(int pid);
   int set_wrr_scheduler(int pid, int weight);
   ```
   By default, Linux assigns a newly created process the completely fair scheduling (CFS) class. The system call `set_wrr_scheduler()` will change the process `pid`’s scheduling class to our newly implemented WRR policy and assign a weight to the process.

3. Write a user-level test program. The program first forks multiple processes and issues the `set_wrr_scheduler()` system call to configure these processes to be scheduled under the WRR policy with different weights. To verify that these processes are running with WRR, the test program issues the `get_wrr_weight()` system call to query individual processes’ weights. At last, the program set the CPU affinity of these processes (using `sched_setaffinity()`) to the same core so as to test if the WRR policy works.

4. **Bonus task** Extend the WRR scheduler to enforce fairness for process groups. A process group contains multiple processes from the same user. The new group fair scheduler ensures that no matter what are the weights inside a group, the CPU time allocated to individual groups should be the same.

Point distribution

1. The new scheduler is integrated into the Linux kernel and is able to assign the WRR policy to individual processes (60 pts)
2. The new scheduler can run individual processes (e.g., `top` displays that your WRR processes are running), but the CPU shares of processes might not be right (20 pts)
3. The new scheduler proportionally assigns CPU time to individual processes based on their weight (e.g., `top` displays CPU usage proportional to process weights) (20 pts)
4. Group fair scheduler assigns equal CPU time to process groups. (25 pts, **required for 3-member groups**)

Hints
1. I have created a skeleton (kernel/sched_wrr.c) to help you implement the new scheduler. The skeleton contains the functions that might need to be implemented by the new scheduler. The minimum set of functions need to be implemented are enqueue_task_wrr, dequeue_task_wrr, pick_next_task_wrr, put_prev_task_wrr, and task_tick_wrr.

2. Actually, the real-time scheduler (kernel/sched_rt.c) has implemented a similar weighted round robin policy for the SCHED_RR class. You can get ideas from the implementation of this scheduler. Specifically, the real-time scheduler implements multiple queues with different priorities. In this project, we just need to implement a single queue. Structure rt_prio_array contains the multi-priority queue. Starting from here, you will find out how the runqueue is initialized and how an entry (e.g., a task) is added or deleted from the queue.

3. The main scheduling function in Linux is schedule() in kernel/sched.c. This function calls policy specific pick_next_task() to select next task to run. It is a good starting point to understand how a CPU scheduler works.

4. To implement a round robin policy, timers are needed to keep track the execution of individual processes. If one process finishes its time slice, another process needs to be scheduled. Whenever a timer interrupt occurs, the function scheduler_tick() in kernel/sched.c will be called. It tests if the current running process has used up all its time slice. If so, the main scheduling function schedule() will be called to find the next task to run.

5. Necessary changes are needed in some existing source files (e.g., sched.c and sched.h) to implement the new scheduler. Searching keyword “Jia” will lead you to my hints.

6. Refer to Project-1 for how to compile Linux kernel and add system calls.

7. The macro container_of returns a pointer to the parent structure.

8. Use printk to debug. Happy debugging...