1 Objectives

1. Learn about scheduling in Linux kernel
2. Understand the tradeoffs involved in scheduling
3. Work on a codebase of an existing operating system

2 Assignment

In this assignment you will work in the same virtual environment we used in Assignment 1.
You are provided with a patch for Linux kernel 3.18.3 which adds a new dummy scheduler. This scheduler is designed to coexist with the existing Linux real-time and fair schedulers. It schedules the tasks whose kernel priority is in the range [131-135] (which translates to the nice priority range [11-15]), but it still has a lower priority than the fair scheduler. The dummy scheduler implements the first-come-first-served (FCFS) scheduling policy.

Your task is to modify the dummy scheduler to add the following functionality:

1. Priority scheduling with support for 5 priority levels
2. Preemption due to a task of a higher priority becoming available
3. Preemption due to running task’s timeslice expiry
4. A mechanism to prevent the starvation of processes with lower priority

Although the dummy scheduler covers 5 different priority levels, it treats all tasks equally, putting them in a single FIFO queue. You should implement priority scheduling, where the task with the highest priority is chosen to be scheduled. You should only handle the tasks with priorities in the range [11-15] (lower value indicates higher priority). You can set the priority of a task using the setpriority(2) system call. Alternatively you can run a program with the nice(1) utility to set its priority. The renice(1) utility can change the priority of a running task. The CPU should always belong to the ready task of the highest priority. This means that if a task of higher priority than the currently executing one becomes available, you should preempt the running task.

A task with a higher priority can become available for a variety of reasons - a new task is activated, a blocked task is woken up, a lower priority task had its priority changed, etc. You should also handle the case when the running task yields the CPU.

Your scheduler should alternate between tasks of the same priority in a round-robin manner. There is a timeslice parameter in the dummy scheduler code that defines how long a task can execute in a single burst before being preempted. The parameter is in jiffies (a jiffy is one tick of the interrupt timer). The parameter can be set through the /proc filesystem at /proc/sys/kernel/sched_dummy_timeslice. To summarize, the running task should continue executing until either its timeslice expires or a higher priority task becomes available, whichever happens first.

Finally, you should solve the starvation problem that comes with priority scheduling. To prevent lower priority tasks from indefinitely waiting on the CPU, you should implement priority aging. Tasks waiting on the CPU should have their priorities temporarily increased proportionally to the time they spent waiting. The amount of CPU time they get should be proportional to their priority. The age threshold parameter in the dummy scheduler code defines how long a task should wait in the runqueue before its priority gets increased by one level. This parameter is also accessible through the /proc filesystem at /proc/sys/kernel/sched_dummy_age_threshold. The priority of a task should never change to a value outside the [11-15] range due to aging. The scheduler is required to work only for the single CPU case. You do not need to worry about SMT issues like per-CPU runqueues, locking and task migration.
3 Linux Scheduler

The Linux scheduler source code is contained within the kernel/sched directory and the include/linux/sched.h header. The patch adds a new file kernel/sched/dummy.c, where you should implement most of your solution. The skeleton code of relevant functions you need to implement is included there. Depending on your implementation, some of them may remain empty. You may find that you need to add/change parts of code in other files as well. Don’t be afraid to do so. One of the goals of this assignment is also to make you comfortable working on a large code base of an operating system kernel. This means you should read through several source files and understand some parts of those. The two main files that contain the implementation of the modular scheduler are kernel/sched/sched.h and kernel/sched/core.c. The implementations of fair and real-time schedulers are in kernel/sched/rt.c and kernel/sched/fair.c respectively. You may want to take a look at them to see how it implements the various functions that you are also required to implement. You can find out more about the Linux scheduler in [1].

4 Deliverables

This assignment is due on May 28, 6am.

As for the previous assignments, you need to submit a ZIP file containing the following:

1. a .patch file that contains your solution
2. a README file where you briefly describe your solution

Note: The scheduler is a core part of the system so be sure to backup your VM before a reboot. The provided range of priorities is targeted so that you will not have problems booting if there is no major error in the code but once you start coding and debugging this might change. In order not to loose your work, be sure to take a snapshot before a reboot.

References
