Operating Systems Spring 2013
Assignment #2
Scheduler

March 20, 2013

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

You are provided with a patch for Linux kernel 3.8.3 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 [135-139] (which translates to the nice priority range [15-19]). 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:

- Priority scheduling with support for 5 priority levels
- Preemption due to a task of a higher priority becoming available
- Preemption due to running task's timeslice expiry
- 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 [15-19], which is the lowest 5 priority levels (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 ageing. 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 [15-19] range due to ageing.

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 folder 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 least at the real-time scheduler 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 User Mode Linux

User Mode Linux (UML) is a Linux virtual machine that runs on a Linux host operating system. You will develop your solution in a UML environment. Here is a short tutorial explaining how to build and run a UML guest.

1. Get the kernel 3.8.3 sources (latest stable at the time of writing of this document).

   ```
   wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.8.3.tar.xz
   tar xf linux-3.8.3.tar.xz
   ```

2. Patch the kernel sources with the dummy patch.

   ```
   cd linux-3.8.3
   patch -p1 -i /path/to/dummy.patch
   ```

3. Configure and build the kernel. It is a good idea to build to a different output directory, so you have clean sources to diff against.

   ```
   make ARCH=um O=/path/to/build defconfig
   make ARCH=um O=/path/to/build
   ```

4. Get a root file system image. You can find a collection of images here: http://fs.devloop.org.uk/. Pick one compatible with UML.

   ```
   wget http://fs.devloop.org.uk/filesystems/Debian-Wheezy/Debian-Wheezy-AMD64-root_fs.bz2
   bunzip2 Debian-Wheezy-AMD64-root_fs.bz2
   ```

5. Create a swap file (optionally).

   ```
   dd if=/dev/zero of=swap.img count=256 bs=1M
   mkswap swap.img
   ```

6. Finally, start the guest VM.

   ```
   ./linux ubda=/path/to/root_fs ubdb=/path/to/swap.img mem=256M con0=fd:0,fd:1 con=pts
   ```

7. Log in as root, no password needed. You can stop the VM with the `halt` command. You can mount the filesystem of the host inside the guest using (the -o parameter is optional):

   ```
   mount none /mnt/point -t hostfs -o /path/to/host/dir
   ```

After you have successfully set up everything, you can start adding your own code. From now on, you just need to build and run:

```
make ARCH=um O=/path/to/build/dir
./linux ubda=/path/to/root_fs ubdb=/path/to/swap.img mem=256M con0=fd:0,fd:1 con=pts
```

For more information about UML visit
http://user-mode-linux.sourceforge.net/old/UserModeLinux-HOWTO.html.

5 Debugging and Testing

If you just want to output some values from your code, take a look at `printk(9)` and `syslog(2)`. The syntax of printk is `printk("log level" "message", <arguments>);`. Depending on the log level, printk can write to the console, or to the system log. You can see messages logged in the system log using `dmesg` command. Since UML is a task like any other on your machine, you can also debug it using `gdb` like any other task.

For testing, you can use setpriority, nice and renice to set/change task priorities. Your solution will be expected to work correctly with them. You can inspect the running tasks with standard utilities `top(1)` and `ps(1)`. We shall also provide some test cases.

6 Deliverables

You need to submit a patch file named `sched.patch` obtained by running:

```
diff -pruN a b > sched.patch
```

where `a` is the Linux kernel source folder patched by the provided `dummy.patch` and `b` is the Linux kernel source folder with your solution. Starting from unmodified kernel 3.8.3 sources, your solution needs to be obtained by running the sequence of commands:

```
patch -p1 -i /path/to/dummy.patch
patch -p1 -i /path/to/sched.patch
```

If you do not submit a working patch, or if your solution does not compile or cannot be started, you will not get any points. Therefore, it is very important that you try to patch unmodified source as above, compile and run it successfully before submitting your patch.

The deadline is April 10, 2013 23:59 CET. This is a hard deadline.

References

[1] Wolfgang Mauerer, Professional Linux Kernel Architecture. Wrox; 1st edition (2008). 3