Homework 2 Wet

Due date: 04/05/2016 12:30

Teaching assistant in charge:
- Assaf Rosenbaum

Important: the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:
- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You’re not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding hw2, put them in the hw2 folder

Only Arthur, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly.
Introduction

As we have seen, handling many processes in pseudo parallel way may have many advantages, but it also comes with a cost. The kernel must keep a track of the relevant data for many processes, and context switching itself is an operation that consumes some resources. Thus, it makes sense, for important short CPU bounded processes, to let them run without interruption and avoid unnecessary context-switches to the I/O bounded processes.

In this assignment, you will add a new scheduling policy to the Linux kernel. The new policy, called SCHED_SHORT, is designed to support important short CPU bounded processes and will schedule some of the processes running in the system according to a different scheduling algorithm that you will implement. Because predicting the exact time a task will take is difficult, sometimes, short processes will take a bit longer than expected we would like to provide a mechanism to handle such cases.

1. Detailed Description

Your goal is to change the Linux scheduling algorithm, to support the new scheduling policy. A process that is using this policy will be called a SHORT-process.

Only an OTHER-process (with SCHED-OTHER policy) might be converted into a SHORT-process. This is done by the sched_setscheduler() system call. When the policy of a process is set to SCHED_SHORT, the caller of the sched_setscheduler() should also inform the operating system of the requested time and number of cool off cycles for the process. The requested time is given in milliseconds and can be from 1 to 3000, the number of cool off cycles must be an integer between 0 and 5. The SHORT-process will first get a time-slice equals to requested time, if the SHORT-process did not finish its run during this time-slice it is sent to a cool off period, a period in which he considered overdue and gets low priority, after finishing the cool off period, the process gets a new time slice equals to requested time. The cool off period is equal to the process requested time. This is called a cool off cycle.

A SHORT-process that used all of its cool off cycles and didn't finish will remain overdue until the end of its run. SHORT-process can never be changed back into an OTHER-process (or a real-time process). Once a process has become SHORT, it will remain SHORT (might be Overdue-SHORT), until it exits.

You as the kernel designers may decide to maintain any other kernel data-fields needed for a SHORT-process.
Scheduling policies order

Any SHORT-process that is not overdue, will receive higher priority than the OTHER-processes. However, an Overdue-SHORT-process will receive the lowest priority in the system.

So the scheduler must run the processes in the system in this order:

1. Real time (FIFO and RR) processes
2. SHORT-Processes
3. OTHER processes
4. Overdue-SHORT Processes
5. The idle task

Therefore, the new scheduler should ignore SHORT-processes as long as there are real-time ready-to-run processes in the system, and ignore Overdue-SHORT-processes while there are any OTHER ready-to-run processes (also in expired (!) priority queue). In general, SCHED_OTHER and SCHED_SHORT scheduling policies are different and not related policies. For example, SHORT-process can never move to expired priority queue, which is related only to SCHED_OTHER scheduling policy.

An important note!
While developing, it is strongly recommended that you will give the OTHER-processes a higher priority than the SHORT-processes, and only at the end, when you are convinced that it works properly, change it and give the SHORT-processes the higher priority. So, while developing, if a SHORT-process is running and an OTHER-process wakes up, the SHORT-process should be switched off. After you are convinced the scheduling mechanism works well, you should change it to the way it should be – that a SHORT-process doesn't give up the processor for a regular process.
The reason for this is that if you run the system with the SHORT-processes priority higher than the SCHED_OTHER priority, and you have a bug that contains an infinite loop or something like that, than you won't be able to stop the system anyway but crashing it, since the OTHER-processes will not be scheduled, including in their kernel mode.

Scheduling SHORT-processes

The CPU should be given to the ready-to-run SHORT-process $p$ that has the highest priority (but is not overdue). The priority is the static priority given to $p$ on its
creation, 120+-nice as for any OTHER- and SHORT-processes. After that, \( p \) runs in FIFO with other SHORT-processes having same priority.

As stated above, between SHORT-processes, the one with the higher priority (minimal priority number) should get the CPU, let’s call it \( p \). You should not switch \( p \) for another (may be new) SHORT-process that has same priority, till the end of \( p \)'s time-slice. However, if another SHORT-process with higher priority appears in the run_queue, the higher-priority SHORT-process should get the CPU.

Thus, a SHORT-process might be removed (switched off) from the CPU in the following cases:

1. A real time process returned from waiting and is ready to run.
2. Another SHORT-process returned from waiting or from a cool off period, and it has higher priority.
3. The SHORT-process forked, and created a child (see explanation in the next section).
4. The SHORT-process goes out for waiting.
5. The SHORT-process ended.
6. The SHORT-process yields the CPU.
7. The SHORT-process finished its time-slice and becomes overdue
8. The nice() call has changed the priority of some lower priority SHORT-process to have higher priority

In any case that a SHORT-process has left the CPU without finishing its time slice you should remember the remained part, the process will use it in the next time it is chosen to run.

**Forking a SHORT-process**

i. The policy of the child is SCHED_SHORT.
ii. The child’s static priority is the same as of its parent.
iii. The parent gives up the CPU and the next task to run is the child process.
iv. Child’s number of cool off periods = \( \left\lceil \frac{\text{ParentNumberOfCoolOffCycles}}{2} \right\rceil \)
   
   New parent’s number of cool off periods = \( \left\lceil \frac{\text{ParentNumberOfCoolOffCycles}}{2} \right\rceil \)

v. Child’s initial time-slice = \( \left\lceil \frac{\text{ParentRemainingTime}}{2} \right\rceil \)
   
   New parent’s remaining time = \( \left\lfloor \frac{\text{ParentRemainingTime}}{2} \right\rfloor \)

**Scheduling Overdue-SHORT-processes**

Overdue-SHORT-processes do not consider their priority, as if they all have the same priority. We can imagine a queue of ready-to-run Overdue-SHORT-processes waiting for CPU. Among the Overdue-SHORT-processes, the CPU should be given
to the ready-to-run process that is waiting for longest time. That is actually FIFO
scheduling. Returning from a waiting is a new entrance to the queue and returning
process needs to wait again to get to the head of the queue. The chosen Overdue-
SHORT-process should run until it finishes or goes to wait. Of course any other
scheduling has a higher priority than Overdue-SHORT.

For summary, an Overdue-SHORT-process might be switched off from the CPU in
one of the following cases:

1. A higher priority policy process returned from waiting.
2. The process goes out for waiting or yields the CPU.
3. The process ended.

**Forking an Overdue-SHORT-process**

When an Overdue-SHORT-process is forking, the child is also Overdue-SHORT and
it enters the Overdue-SHORT ready-to-run queue and waits for its turn to run. The
only other difference between forking an overdue SHORT process to a regular
SHORT process is in section (v):

Child’s remaining cool off time time-slice = New parent’s remaining cool off time =
Parent remaining cool off time (before the fork)

**2. Technicalities**

**New policy**

You should define a new scheduling policy `SCHED_SHORT` with the value of 5 (in
the same place where the existing policies are defined).

Upon changing the policy to `SCHED_SHORT` using `sched_setscheduler()`, all of
your algorithm-specific variables and data structures should be initialized/updated.
If the number of trials was an illegal value, -1 should be returned, and you should
set ERRNO to EINVAL.
In other cases you should retain the semantics of the `sched_setscheduler()`
regarding the return value, i.e., when to return a non-negative value and when -1.
Read the man pages for the full explanation.

Things to note:

- A process can change the scheduling policy of another process.
  Make sure that the user can change the policy for all his processes, and root
can change the policy for all processes in the system, but neither user nor
root can change the policy of a SHORT-process, "Operation not permitted"
error should return.
- The system calls `sched_{get,set}_scheduler()` and `sched_{get,set}_param()`
  should operate both on the OTHER-processes (as they do now) and on
SHORT-processes, but, again, remember that a SHORT-process cannot be
changed into a different policy (but may become overdue).
Policy Parameters

The `sched_setscheduler()`, `sched_getparam()` and `sched_setparam()` syscalls receive an argument of type `struct sched_param*`, that contains the parameters of the algorithm. In the current implementation, the only parameter is `sched_priority`. The SCHED_SHORT algorithm must extend this struct to contain other parameter of the algorithm.

```c
struct sched_param {
    int sched_priority; // ignored for SHORT processes
    int requested_time;
    int numer_of_cooloffs;
};
```

When `sched_setscheduler()` is invoked for SCHED_SHORT it should not change the process priority.

Notice that process cannot become Overdue-SHORT via the above system calls, a SHORT-process turns overdue only as a result of long run.

It should be impossible for any user (or root) to change the number of cool off cycles of any SHORT-process to a different value than initial.

Calling `sched_setscheduler()` on a SHORT process should always fail (EPREM error code)

It is allowed to change the requested time for a SHORT process, using `sched_setparam()`, the effect will take place only if the process gets a new time slice, e.g. let’s say we have a SHORT process with requested time 2000ms and 2 cool off cycles. Assuming this process is asking for new requested time of 3000ms, it will finish the 2000ms time slice, 2000ms cool off period and then get a 3000ms time slice and so on.

Querying system call

Define the following system call to query a process for being SHORT:

- syscall number 243:
  ```c
  int is_SHORT(int pid)
  ```
  The wrapper will return 1 if the given process is a SHORT-process, or 0 if it is already overdue.

- syscall number 244:
  ```c
  int remaining_time(int pid)
  ```
  The wrapper will return the time left for the process at the current time slice. For an overdue process, it should return the reminded cool off time.
• syscall number 245:

```c
int remaining_cooloffs(int pid)
```

The wrapper will return the number of cool off periods left for the SHORT process. Whenever a cool off period has started it is no longer consider as a remaining cool off period. For example, let p be a SHORT process with requested time 100ms and 1 cool off period. After running for 100ms, it will go to a cool off period, during that time remaining_cooloffs should return 0.

In case of an unsuccessful call, wrappers should return -1 and update errno accordingly, like any other system call. In case the process is not a SHORT nor Overdue-SHORT-process update errno to EINVAL.

Note that the wrapper for these system call should use the interrupt 128 to invoke the system_call() method in the kernel mode, like regular system calls.

### Scheduling

Update the necessary functions to implement the new scheduling algorithm.

Note that:

- You must support forking a SHORT- and Overdue-SHORT-process as defined above.
- You should not change the function context_switch or the context switch code in the schedule function.
- When a higher-priority (according to all of the rules defined above) process is waking up, it should be given the CPU immediately.

### 3. Important Notes and Tips

• Reread the tutorial about scheduling and make sure you understand the relationship between the scheduler, its helper functions, the run_queue, waitqueues and context switching.

• Think and **plan** before you start – what will you change? What will be the role of each existing field or data structure in the new (combined) algorithm?

• Notice that it is dangerous to make the processes priority above all OTHER processes. When testing it you can easy run in the problematic situations when your kernel is not booting. Thus first set the priority of other processes higher than SHORT-processes and test them well and only after that do the switch the priorities to how it should be.

• **Note that allocating memory** `(kmalloc(buf_size, GFP_KERNEL) and kfree(buf))` from the scheduler code is dangerous, because kmalloc may sleep. This exercise can be done without dynamically allocating memory.

• You must **not** use recursion in kernel. The kernel uses a small bounded stack (8KB), thus recursion is out of question. Luckily, you don’t need recursion.
• Your solution should be implemented on kernel version 2.4.18-14 as included in RedHat Linux 8.0.

• You should test your new scheduler very thoroughly, including every aspect of the scheduler. There are no specific requirements about the tests, inputs and outputs of your thorough test and you should not submit it, but you are very encouraged to do this.

• We are going to check for kernel oops (errors that don’t prevent the kernel from continue running such as NULL dereference in syscall implementation). You should not have any.
  If there was kernel oops, you can see it in dmesg (dmesg it’s the command that prints the kernel messages, e.g. printk, to the screen).
  To read it in more conveniently: **dmesg | less -S**

4. Submission

You should create a zip file (**use zip only**, not gzip, tar, rar, 7z or anything else) containing the following files:

a. A tarball named **kernel.tar.gz** containing all the files in the kernel that you created or modified (including any source, assembly or makefile).

  To create the tarball, run (inside VMWare):

  ```
  cd /usr/src/linux-2.4.18-14custom
  tar -czf kernel.tar.gz <list of modified or added files>
  ```

  Make sure you don't forget any file and that you use **relative** paths in the tar command, i.e., use kernel/sched.c and not /usr/src/linux-2.4.18-14custom/kernel/sched.c

  Test your tarball on a "clean" version of the kernel – to make sure you didn’t forget any file.

  If you missed a file and because of this, the exercise is not working, you will get 0 and resubmission will cost 10 points. In case you missed an important file (such as the file with all you logic) we may not accept it at all. In order to prevent it you should open the tar on your host machine and see that the files are structured as they supposed to be in the source directory. It is highly
recommended to create another clean copy of the guest machine and open the tar there and see it behave as you expected.

To open the tar:

```
cd /usr/src/linux-2.4.18-14custom
tar -xzf <path to tarball>/kernel.tar.gz
```

b. A file named `hw2_syscalls.h` containing the syscalls wrappers.

c. A file named `submitters.txt` which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890
Ken Thompson ken@belllabs.com 345678901
```

**Important Note:** Make the outlined zip structure **exactly**. In particular, the zip should contain only the 3 files, without directories.

You can create the zip by running (inside VMware):

```
zip final.zip kernel.tar.gz hw2_syscalls.h submitters.txt
```

The zip should look as follows:

```
zipfile +-                        
    |                           
    +- kernel.tar.gz           
    |                           
    +- submitters.txt          
    |                           
    +- hw2_syscalls.h
```

*Have a Successful Journey,*

The course staff