Homework 2 Wet

Due Date: 08/12/2016 23:00

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 Arie, 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 track of the relevant data for many processes, and context switching itself is an operation that consumes some resources. Thus, it makes sense to let important short CPU bounded processes to 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 need to run a bit longer than expected. We would like to provide a mechanism to handle such cases. There are many techniques to handle run time misprediction. In this assignment we'll take the approach of punishing a process for exceeding its predicted run time by giving it the lowest priority in the system for some time.

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 should also inform the operating system of the requested time for the process. The requested time is given in milliseconds and can be from 1 to 3000. The SHORT process will get a time slice equals to requested time, if the SHORT process did not finish its run during this time slice it becomes overdue and gets very low priority. After running for some time with low priority the process policy is changed back to SCHED_OTHER. 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:

- Real time (FIFO and RR) processes
- SHORT processes
- OTHER processes
- Overdue SHORT processes
- 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 by 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 that has the highest priority (but is not overdue). The priority is the static priority given to the process on its creation (120 + nice, like OTHER processes). Between SHORT processes with the same priority the order is FIFO. You should not switch a SHORT process for another (may be new) SHORT process that has same priority. 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:

- A real time process returned from waiting and is ready to run.
- Another SHORT process returned from waiting, and it has higher priority.
- The SHORT process forked, and created a child (see explanation in the next section).
- The SHORT process goes out for waiting.
- The SHORT process ended.
- The SHORT process yields the CPU.
- The SHORT process finished its time slice and becomes overdue.
- 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

1. The policy of the child is SCHED_SHORT.
2. The child’s static priority is the same as that of its parent.
3. The parent gives up the CPU and gets transferred to the end of the queue. The child is placed at the beginning of the queue.
4. Child’s time slice = \([\frac{ParentTimeSlice}{2}] + ParentTimeSilce \times \%2\)
5. New parent’s time slice = \([\frac{ParentTimeSlice}{2}]\)

(note - \([x]\) represent the integer part of \(x\))

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:

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

After a process has run as an overdue SHORT for a time equals to twice of its requested time its policy is changed back to SCHED_OTHER. You should maintain the sleep_avg field for SHORT process in the same manner it’s maintained for OTHER processes. The dynamic priority (prio) of a process changing back to OTHER after finishing its time as an overdue SHORT should be determined at the time of the change. Any OTHER process may be changed to SHORT, regardless if it was SHORT in the past.

Forking an Overdue SHORT Process

When an overdue SHORT process is forking, the child is also an overdue SHORT and it enters the overdue SHORT ready to run queue (at the end) and waits for its turn to run. The child will need to run as an overdue SHORT for the same amount of time left for its parent on the time of the fork.
A Simple Example

Let's assume we have a SHORT process created with requested time of 50ms. If this process runs for 50ms it becomes overdue. After running for 100ms as an overdue SHORT the process policy is changed back to SCHED_OTHER. Now, lets further assume that after running for 75ms as an overdue SHORT the process forks. Both the parent and the child will need to run 25ms as overdue SHORT before their policy will be changed to SCHED_OTHER.

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 requested time 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. In this case you should return EPERM.

- The system calls sched_getscheduler() and sched_getparam() should operate both on the OTHER processes (as they do now) and on SHORT processes, but, remember that a SHORT process cannot be changed into a different policy directly.

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;
};
```
When sched_setscheduler() is invoked for SCHED_SHORT it should not change the process priority. Any attempt to change the policy of a SHORT process should fail (EPERM error code). It is allowed to change the requested time for a SHORT process. However, it is forbidden to decrease the requested time to a value smaller than the actual time the process has already ran or to a value higher than 3000 ms (EINVAL error code). Keep in mind that you must maintain backward compatibility. If a program was written prior to your changes and could be executed on the system, it should still run after your changes.

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

**syscall number 243:**

```c
int is_short(pid_t 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 short_remaining_time(pid_t pid)
```

For a regular SHORT process, the wrapper will return the time (in ms) left before it becomes overdue. For an overdue process, it should return the time left before the process policy will be changed to SCHED_OTHER.

**syscall number 245:**

```c
int was_short(pid_t pid)
```

If the process policy is SHORT, EINVAL error code should be returned. Otherwise the wrapper should return 1 if the process was a SHORT process before and 0 if not.

In case of an unsuccessful call, wrappers should return -1 and update errno accordingly, like any other system call. For the first and second system calls, in case the process is not a SHORT process, EINVAL error code should be returned. 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. If there is no specific error code defined, chose one as you see fit. For example, if no process with PID pid exists ESRCH makes the most sense. We won’t deduct points for choosing the “wrong” error code (but it will make your life harder when you’ll debug the system).

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.

Important Notes and Tips

- All time related values to/from the user are in ms, but the kernel measure time in jiffies. Don’t forget to convert (look for existing conversion kernel code).
- Reread the tutorial on scheduling and make sure you understand the relationship between the scheduler, its helper functions, the run_queue, wait queues 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 SHORT processes’ priority higher than all OTHER processes. When testing it you can easily run into the problematic situations where 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 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 the kernel. The kernel uses a small bounded stack (8KB), thus recursion is out of the 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, nor the inputs and outputs of your thorough tests, and you should not submit them, but you are very encouraged to test thoroughly.
- We are going to check for kernel oops (errors that don’t prevent the kernel from continuing to run, such as NULL dereference in syscall implementation). You should not have any. If there was a kernel oops, you can see it in dmesg (dmesg is the command that prints the kernel messages, e.g. printk, to the screen). To read it more conveniently: dmesg | less -S
- During your work you might encounter some small kernel bugs, you are not supposed to fix them, but make sure your code meets the assignment requirements. For example, in your kernel version, changing the static priority of a task (using the nice() system call) doesn’t cause a context switch. This might cause the process to run while there’s a task with a higher priority in the run_queue. You don’t need to fix this bug (you may if you
want to), but you need to make sure that when changing the static priority of a SHORT
process a context will happen if needed.

- If something is not defined in the assignment, it will not be tested. You may implement in
any way you see fit. For example, the error code to return if the pid provided to a system
calls can't be found is not defined. Chose an error code that you like, ESRCH will be the
reasonable choice.

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):

```bash
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.
For example, 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 your 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:

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

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

**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):

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

The zip should look as follows:

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

Have a Successful Journey,
The course staff