Homework 2 Wet

Due date: 3.12.2015, 12:30
Teaching assistant in charge: Assaf Rosenbaum

Important: the Q&A for the exercise will be handled via Piazza forum only (emails will not be answered). 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 your 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

Postponements can be authorized only by Arthur, the TA in charge. In case you need a postponement contact him directly.

1 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.
2 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 importance of the process. The requested time is given in milliseconds and can be from 1 to 3000, the importance must be an integer between 0 and 75. The SHORT-process will first get a time slice equals to the requested time, if the SHORT-process didn’t finish his run during this time slice it will get a new time slice with probability \( p = \frac{\text{importance}}{100} \), in case a new time slice is assigned to the process it will be: \( \text{NewTimeSlice} = \lceil p \cdot \text{PrevTimeSlice} \rceil \).

A SHORT-process that finished his time slice and was not assigned with a new one, will be considered an Overdue-SHORT-process. A SHORT-process or Overdue-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.

For example, let us assume that a short process was created with a requested time 256 and importance 50. After running for 256 milliseconds the process has exercised his time slice, now he will be assigned with a new time slice with probability \( p = \frac{1}{2} \) and with the same probability will become overdue, in case he will get a new time slice it will be of 128 milliseconds. Now assuming the process was assigned with a new time slice and finished it as well, he will get a new time slice of 64 milliseconds with probability \( \frac{1}{2} \) and become overdue with the same probability and so on...

2.1 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 SCHEDHORT 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 tip!
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.

2.2 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 A on its creation, 120+/nice as for any OTHER-processes. After that, A runs in Round-Robin (RR) with other SHORT-processes having same priority. A can participate in RR only if it was assigned with a new time slice, a SHORT-process that has finished his time slice and was not assigned with a new one is considered overdue, and should not be selected. For this case, you should make the necessary changes to your data structures to start treat it as an Overdue-SHORT-process (and choose a different process to run accordingly). As stated above, between SHORT-processes, the one with the higher priority (minimal priority number) should get the CPU, let’s call it B. You should not switch B for another (may be new) SHORT-process that has same priority, till the end of B’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, 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 his time slice
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 his time slice, you should remember the remained part of his time slice and use it the next time he is selected to run.

2.3 Forking a SHORT-process
1. The policy of the child is SCHED_SHORT.
2. The parent gives up the CPU and placed at the back of the line, the child gets the CPU.
3. The child’s static priority is the same as of his parent.
4. The child’s importance is the same as of his parent.
5. Child’s initial time slice (the one after the fork) = ⌈\(\frac{\text{ParentRemainingTime}}{2}\)⌉
6. New parent’s remaining time = \(\max\{\lfloor\frac{\text{ParentRemainingTime}}{2}\rfloor, 1\}\)

Later the time slices are calculated by the original requested time. For example: let us assume a SHORT process was created with requested time of 256 milliseconds and importance 50. If after 56 milliseconds he creates a child they each has a time slice of 100 milliseconds, the parent is switched off from the CPU and moving to the back of the line and the child gets the CPU. If any of them is lucky enough to get a new time slice it will be 128 milliseconds, the next one will be 64 milliseconds and so on.

2.4 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.
2.5 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. Unlike not overdue SHORT processes, here the parent is not switched off from the CPU.

3 Technicalities

3.1 New policy

You should define a new scheduling policy SCHED_SHORT with the value of 10 (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 any of the parameters has an illegal value you should return -1, and 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.

- Overdue SHORT is a state of a SHORT process and not a different policy.

3.2 Policy Parameters

The sched_setscheduler(), sched_getscheduler(), sched_setparam() and sched_getparam() syscalls which already implemented in your kernel, 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;
    int requested_time;
    int importance;
};
```
When sched_setscheduler() is invoked for SCHED_SHORT it should not change the process priority.
Notice that sched_priority might have different meaning than you think so make sure you understand it, when changing a process policy to SHORT this parameter should be ignored.

A call for sched_setparam() on an overdue SHORT process should always fail. For a non overdue SHORT process, the only parameter change allowed is a decrease of his importance, the other fields of the struct are ignored.

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

- syscall number 243:

  ```c
  int is_overdue_SHORT(pid_t pid)
  ```

  The wrapper will return 1 if the given process is an overdue SHORT-process, or 0 if it is not.

- syscall number 244:

  ```c
  int remaining_time(pid_t pid)
  ```

  The wrapper will return the time left for the process at the current time slice, for an overdue process it should return 0.

- syscall number 245:

  ```c
  int number_of_time_slices(pid_t pid)
  ```

  The wrapper will return the number of time slices assigned to the SHORT process, including the one assigned when the SHORT process was created. Running as an overdue process doesn’t consider as an assignment of a time slice.

  For example, let us assume a SHORT process was created with a requested time of 128 milliseconds. During this time slice, number_of_time_slices should return 1, if the process has finished his time slice and was assigned with a new one the answer will now be 2. Now assuming that the process has finished his new time slice and become overdue, all subsequent calls to number_of_time_slices will return 2.

  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 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.

3.4 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.

4 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 easily 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 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.
- You may assume the system has only one CPU (!!!) but as you saw on the tutorial even in that case some synchronization is required. Make sure that relevant parts of your code enjoy this synchronization protection as well.
• 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.

• Focus your attention on scheduling code and logic, this is the main part of this exercise. Don’t waste your time on trying to find any possible corner case of parameters check for the system calls, just make sure you check for the cases we’ve defined. Our tests will do the same...

• Whenever an error value is not defined you may choose any error code you see fit.

5 Submission

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

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

2. A file named hw2_syscalls.h containing the syscalls wrappers.

3. 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