Homework 2 Wet

Due date: 08/05/2015 12:30

Teaching assistant in charge:

- Assaf Rosenbaum

Important: the Q&A for the exercise will take place at a public forum Piazza only. Please note the forum is a part of the exercise, clarifications/corrections that will be published 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

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.

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 trials** of the process. The requested time is given in milliseconds and can be from 1 to 5000, the number of trials must be an integer between 1 and 50. The SHORT-process will first get a time-slice equals to requested time, but SHORT-process could try to finish its run in number of trials, each in the length of its time-slice (that is different for each trial). The SHORT-process's time-slice is calculated at the beginning of each trial *n* according to the following function:

$$TimeSlice_n = \left\lfloor \frac{RequestedTime}{n} \right\rfloor$$

A SHORT-process that used all of its trials or that his next time slice equals to 0 and didn’t finish 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.

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 (<u>also in expired (!) priority queue</u>). 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 Round-Robin (RR) with other SHORT-processes having same priority. p can participate in RR only its "*number of trials*" times.

A SHORT-process that has (exactly) 0 remaining trials left 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 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 $\mathrm{run\_queue}$, 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 this time-slice (it may return to another trial)
8. The SHORT-process finished this time-slice and it was its last trial (it has more code to do). The process became an overdue process.
9. 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 its time-slice and use it later – counting all the time-slice usage <u>as a single trial</u>.

Note that if there is only one SHORT-process for some priority, it simply gets "number of trials" time-slices with no interruptions.

<u>**Forking a SHORT-process**</u>

i. The policy of the child is SCHED_SHORT.
ii. The parent gives up the CPU and the scheduler goes to the next task according to the SCHED_SHORT scheduler.
iii. The child's static priority is the same as of its parent.
iv. Child's number of trials = $\left\lceil \frac{ParentNumberOfTrails}{2} \right\rceil$
New parent's number of trials = $\left\lfloor \frac{ParentNumberOfTrails}{2} \right\rfloor$

v. Child's initial time-slice (for its first trial) = $\left\lceil \frac{ParentRemainingTime}{2} \right\rceil$
New parent's remaining time = $\left\lfloor \frac{ParentRemainingTime}{2} \right\rfloor$
Later the time slices are calculated for each trial as explained above (by the requested time).

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.

Complexity requirements
The space complexity of the scheduling process should remain O(*n*), when *n* is the number of processes in the system.
Regarding the time complexity, you should make an effort to have the scheduler as fast as possible. Achieving O(1) time complexity for **every** possible choice of a process. Specifically, when a process exit or goes out for wait, or when a SHORT-process becomes overdue, or when an OTHER-process finishes its time slice, the next process to run must be chosen in O(1).

2. Technicalities

New policy

You should define a new scheduling policy `SCHED_SHORT` with the value of 4 (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.

```
struct sched_param {
   int requested_time;
   int trial_num;
};
```

When a process is turning to be SHORT, initialize `sched_param->sched_priority` to zero. Anyway while `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 trials of any SHORT-process to a different value than initial.

Querying system call

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

- syscall number 243:
  ```
  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:
  ```
  int remaining_time(int pid)
  ```
- The wrapper will return the time left for the process at the current time slice, for overdue process it should return 0.
- syscall number 245:
  ```
  int remaining_trails(int pid)
  ```
  The wrapper will return the number of trails left for the SHORT process, for overdue process it 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. Testing

Monitoring the scheduler

To measure the impact of your scheduler, and to see that it really works, you should add a mechanism that collects statistics on your scheduler behavior. For every creation of a process - not only a SHORT-process - your mechanism should remember the next 30 process switching events that occurred after the creation, and also the same information after a process ends (process ends on calling do_exit() function). So actually on process creation or ending you should start monitoring the next 30 task switching event.

From these, remember only the 150 latest task switching events. If during 30 events a process is created or ends then you should record 30 events from the new creation/ending. For example, if a process was created and then after 11 scheduling events another process was created then you have to record in total 41 events.

For each task switching it should have the following information:

i. The next task pid and policy.
ii. The previous (current) task pid and policy.
iii. Time of switching (value of jiffies).
iv. Reason of switching should be one of the following:
(if more than one reason is correct, choose the first reason from the list)
    1. A task was created.
    2. A task ended.
    3. A task yields the CPU.
    4. A SHORT-process became overdue.
    5. The previous task goes out for waiting.
    6. A task with higher priority returns from waiting.
    7. The time slice of the previous task has ended.

Pay attention that by using a nice() call or by changing the policy to SHORT, a task can get a higher priority than a current task. For such context switch use reason number 6: "A task with higher priority returns from waiting".

Define in your kernel and user mode a struct that contain the information on a task switching:

```
struct switch_info
{
int previous_pid;
int next_pid;
int previous_policy;
int next_policy;
unsigned long time;
int reason;
};
```

Monitoring one task switching should be O(1).

As a rule the context-switch is defined to happen whenever context_switch() is called. As you can see in the kernel code, it can happen only when prev != next. In case the processes are the same there is no context switch and there is no need to log.

- Add a new system call (+ wrapper), with the following prototype:

```
int get_scheduling_statistic(struct switch_info *);
```

Give get_scheduling_statistic() syscall number 246. This system call gets a pointer to a switch_info array of size 150 in user mode.
The system call should fill the array and return the number of elements that were filled, or -1 (and updating errno accordingly) on error. The memory allocation is done by the user.
Hint: Like in set/get_sched which use `copy_to_user` and `copy_from_user` to copy data from kernel space to user space and vice versa, here you also have to use the function `copy_to_user()`. Use Google, the man pages and the source code to find information about these functions.

Testing program

Write a program that invokes tasks (call it sched_tester.c). The policy of all the tasks (created by this program) should be set to SCHED_SHORT.
They should all do a recursive calculation of a fibonaci number.

(That looks something like this :

```
int fibonaci(int n)
{
   if (n < 2)
      return n;
   return fibonaci(n-1) + fibonaci(n-2);
}
)
```

They needn't return the result value – just do the calculation.

When all tasks are done (use wait()) the program should call get_scheduling_statistic() ,print the output in a clear format (together with the PID's and parameters of the tasks) and finish.

The program should get as arguments pairs of integers:

sched_tester <number_of_trials$_1$> <n_1> <number_of_trials$_2$> <n_2>…

When < number_of_trials $_i$> and <n_i> and is the number of trials for the ith process, and the fibonaci number it is asked to calculate.

Run this program on the following inputs:
i. sched_tester 0 10 100 5 101 43 50 10
ii. sched_tester 99 3 1 5 50 7
iii. sched_tester 98 10 97 20 2 25 3 30 100 35 4 40 5 45

Explain the results. How does the algorithm express itself in the result? The explanation should be submitted as part of the printed submission.

4. Important Notes and Tips

- You may assume that a SCHED_SHORT process will never yield the CPU.
- 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.
- You may assume the system has only one CPU (!!!) but still you might need some synchronization, when editing the kernel data-structures.
- 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.

5. Submission

The submission of this assignment has two parts:

An electronic 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

b. A file named **hw2_syscalls.h** containing the syscalls wrappers.
c. A file named **sched_tester.c** containing your test program.
d. A file named **tester_results.txt** containing the output of the sched_tester and to explanations to it.
e. A file named **submitters.txt** which includes the ID, name and email of the participating students. The following format should be used:

```
Bill Gates bill@t2.technion.ac.il 123456789
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 5 files, without directories.

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

```
zip final.zip kernel.tar.gz sched_tester.c hw2_syscalls.h tester_results.txt
submitters.txt
```

The zip should look as follows:

```
zipfile -+
         |
         +- kernel.tar.gz
         |
         +- submitters.txt
         |
         +- sched_tester.c
```

```
          |
          +- tester_results.txt
          |
          +- hw2_syscalls.h
```

A printed submission

The **printed** submission should contain an explanation of the changes you have made and describe your solution, in particular:

- What data structures did you use
- How were you able to meet the space & time complexity requirements
- Why is your solution correct, i.e. why does it meet the policy definitions.

This part will have a weight of 15% of your grade

Do not print the electronically-submitted source code.

Handwritten assignments will not be accepted.

Have a Successful Journey,
The course staff