Homework 2 Wet

Due Date: 21/05/2017 23:00

Teaching assistant in charge:

- Alon Shtaierman

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 run without interruption and avoid unnecessary context switches to the I/O bounded processes (תהליכים אינטראקטיבים). 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 the
remainder of its run time.

Detailed Description

Your goal is to upgrade 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, and its
priority - which is an integer between 0 and 139 (both these values need to be passed as an
argument to sched_setscheduler(), consider altering struct sched_param in sched.h for this
purpose). 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 until it ends. 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 processes ready to run, even if the only ones in the run_queue are in the expired priority array (These processes are still TASK_RUNNING!). 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 used only for the 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 upon turning its policy into SCHED_SHORT (between 0 and 139). Just as with OTHER processes, the priority is sorted from highest (0) to lowest (139). Between SHORT processes with the same priority the order is FIFO. You should not switch a SHORT process for another SHORT process that has same
priority (for example, when a new SHORT process enters the same priority-queue). However, if another SHORT process with higher priority appears in the run_queue, the higher priority SHORT process should get the CPU as soon as possible. Thus, a SHORT process might be removed (rescheduled) from the CPU in the following cases:

- A real time process returned from waiting and is ready to run.
- Another SHORT process entered the run_queue with a 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 (such as when it left the run_queue in order to go into a wait_queue) 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 child is placed at the end of the priority queue.
4. Child’s time slice = (ParentTimeSlice div 2) + (ParentTimeSilce mod 2)
5. New parent’s time slice = (ParentTimeSlice div 2)

*Note - the division of ParentTimeSlice by 2 represents the integer value of the division (integer division, rounded-down). (E.G.: 5 div 2 = 2, 5 mod 2 = 1)*

Scheduling Overdue SHORT Processes

Overdue SHORT processes all run at the same priority between themselves. We can imagine a queue of ready to run overdue SHORT processes waiting for CPU. All the ready to run overdue
SHORT processes (those who are in the run_queue) share the CPU between each other by way of a Round-Robin scheduling policy. Each overdue SHORT process receives a timeslice equal to $10 \times (140 - \text{short_priority})$ millisecs (meaning every time an overdue SHORT process get its turn to run it receives a timeslice equal to 10 times its designated static priority, a value between 10 and 1400 millisecs, where the highest priority gives the largest timeslice). When the overdue process leaves the run_queue by way of waiting, it retains the remainder of its timeslice at the point of switching for use when it returns. Returning from waiting puts the overdue SHORT process at the end of the current RR cycle (the end of the priority queue), the same applies when the overdue SHORT process yields the CPU. 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:

- The current overdue SHORT process has exceeded its current timeslice
- A higher priority policy process returned from waiting.
- The process goes out for waiting or yields the CPU.
- The process ended.

Forking an Overdue SHORT Process

Since we want to punish SHORT processes which exceeded their run time, any overdue SHORT process which tries to perform a fork should immediately fail. The return value from the do_fork() system call in this case should be -EACCES (meaning the calling process doesn’t have permission to use this system call)

A Simple Example

Let’s assume there are currently only two SHORT processes running in our system - process A with requested time of 100 ms and priority 2, and process B with requested time of 1500 ms and priority 139.

Since process A has the higher priority as a SHORT process, it runs first. Let’s assume A runs
longer than 100 ms, therefore eventually after enough ticks it is rescheduled as an overdue SHORT process with a timeslice of $10 \times (140-2) = 1380$ ms.

Since SHORT processes are always prioritized over overdue SHORT processes, then process A is switched with B.

Now lets assume B also exceeds its requested time: Therefore it too becomes an overdue process with a timeslice of $10 \times (140-139) = 10$ ms.

Now that our system only has overdue SHORT processes, it is A’s turn to run again (A runs first since it entered the overdue queue first). From this point on A and B perform a Round Robin between each other, where every time A gets the CPU it runs for 1380 ms, and every time B gets the CPU it runs for 10 ms. This continues until the processes end.

**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. (This is to make things simpler for you - you can use the uid field in the TASK STRUCT of the calling process for this purpose)

- 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,
remember that a SHORT process cannot be changed into a different policy once it is set as a SHORT process

- Lastly - you will note when working on this exercise that the priority array type (denoted prio_array_t) is widely used in all the scheduling functions. We do recommend you try to incorporate this type into your new design as it will ease your work significantly. You are allowed to use this array even if you plan on using only some of its cells (yes, even if you plan on using only one cell, hint hint).

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;   //between 1 and 3000
    int sched_short_prio; //between 0 and 139
};
```

When sched_setscheduler() is invoked for a process that is SCHED_SHORT it should not change the process priority or requested time. Any attempt to change a SHORT process should return the EPERM error code.

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.
Possible errors:
If no process with the corresponding PID exists - ESRCH
if the given process isn’t a SHORT process (Real-Time or OTHER) - EINVAL

syscall number 244:

```
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 in its current RR timeslice.
Possible errors:
If no process with the corresponding PID exists - ESRCH
if the given process isn’t a SHORT process (Real-Time or OTHER) - EINVAL

syscall number 245:

```
int short_place_in_queue(pid_t pid)
```

For a regular SHORT process (the one designated by the given PID), the wrapper will return the number of SHORT processes that have been scheduled to run before it (not counting itself).
Meaning this system call counts all the SHORT processes that exist at a higher priority than the given process, and those of the same priority that are scheduled before it in its respective priority queue.
For an overdue SHORT process, the wrapper returns the amount of overdue SHORT processes in the overdue queue that are scheduled to run before it.
Note that this call does NOT count any Real-Time processes scheduled before the given process.
Possible errors:
If no process with the corresponding PID exists - ESRCH
if the given process isn’t a SHORT process (Real-Time or OTHER) - EINVAL
Example 1 - We have the following SHORT processes:

If process A calls short_place_in_queue for the PID of process D then the return value should be 3 (counting A, B and C).

Example 2 - if the current process (which is a SHORT process) calls short_place_in_queue with its own PID as the argument, then the return value should (usually) be 0 (obviously - since if the process managed to run that system call then it is currently using the CPU, meaning no other SHORT process should be scheduled before it. A very rare exception to this rule is if a SHORT
process returned from waiting at a higher priority, triggering the need_resched flag, but the system call was invoked before the context switch. In the case of a single processor this should not happen (why?)

In case of an unsuccessful call, wrappers should return -1 and update errno accordingly, like any other system call. For any possible error you can come up with that was not mentioned in the above description you are welcome to consult the possible errno values and return whichever makes the most sense to you. We won’t deduct points for choosing the “wrong” error code in this case (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. The mechanics of switching between two processes should remain as they are - it is only the scheduling algorithm that we are altering, which selects what the next running process needs to be whenever schedule() is invoked.
- When a higher priority (according to all of the rules defined above) process is waking up, it should be given the CPU immediately (I.E. - if the current running process is OTHER, and a SHORT process wakes up from waiting, then need_resched should be activated for the current OTHER process to signal the pending context switch).

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 (as we learned in the tutorials - requested_time * HZ / 1000 is the conversion of requested_time given in ms to the corresponding value in ticks).
- Reread the tutorial on scheduling and make sure you understand the relationship between the scheduler, its helper functions (scheduler_tick, setscheduler, etc.), the run_queue, wait queues and context switching.

- Think and carefully plan your design before you start – what will you change? What will be the role of each existing field or data structure in the new (combined) algorithm? What did the kernel developers do when they wrote the scheduling algorithm for Real-Time and OTHER processes and what can we learn from them?

- 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 for this exercise. The kernel uses a small bounded stack (8KB), thus recursion is out of the question. Luckily, you don’t need recursion. Unlike what you might have learned in other courses, when it comes to kernel-coding - the simple, straightforward solution, is usually better.

- Your solution should be implemented on kernel version 2.4.18-14custom as included in RedHat Linux 8.0.

- You should test your new scheduler very, 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.

- Be aware that processes going to wait might cause a difference in results when running the same test on different virtual systems. Even if your code doesn’t initiate a system call that might cause a wait the process might still be put into a wait_queue due to something called a pagefault (we will learn of this when discussing virtual memory, later on this semester). Our tests are going to focus on the broader aspect of this exercise, such as the correctness of the scheduling algorithm (that the highest priority process does indeed run first), the return values of the given system calls, and of course that the kernel doesn’t crash.
- 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 (meaning little things that might run unlike what you might expect), 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 switch will occur 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.
- Files you should consider changing in this exercise include (but are not limited to):
  - Sched.c, fork.c, exit.c (in the kernel folder)
  - Sched.h (in include/linux/)
  - Entry.s (in arch/i386/kernel)
- Lastly - a lesson many of you might have learned from the first exercise: It is best not to differ your design from the currently available tools in the kernel. Instead, try and understand how the kernel developers built their scheduling algorithm, and aspire to use as many of their data structures and code as you can. For example: prio_array_t which defines the priority array structure used for the Active and Expired arrays in the current run_queue is a convenient way of implementing the new policy (since many of the other function, like active_task, get a prio_array_t as input). Do not be shy to do so even if you do not plan to use all 140 available cells. Understand how structures such as prio_array_t work and how you might employ them in your design to ease your workload.
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. 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:

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

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

```
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