Homework 2 Wet

Due Date: 11/12/2018 23:30

Teaching assistant in charge:

- **Andre Kassis**

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. Some 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 the TA in charge can authorize postponements. In case you need a postponement, please fill out the following form: https://goo.gl/forms/D5nxMxf9Uvgej1SL2

Introduction

As we have seen, handling many processes in a 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. The goal of this assignment is to familiarize you with the Linux scheduler, as it requires you to study it and its mechanisms in depth through changing some of its code and writing some of your own.

You are asked to add a new scheduling policy to the Linux kernel for **normal** (non-realtime) processes: **SCHED_CHANGEABLE**. This policy would go alongside the standard SCHED_OTHER policy that you have learned about in class. We'll denote a process with the SCHED_CHANGEABLE policy an SC process, and a process with the SCHED_OTHER policy an SO process. The changes in the scheduler caused by this new policy will start to be in effect after the newly implemented system call `change()` is invoked with the input parameter of 1.
After change(1) is called (see explanation below), all SC processes will conform to a FIFO order according to their PIDs. Disabling the regime, however, will cause these processes to go back to behaving like OTHER processes.

In-Depth Details

New Scheduling Regime Activation:

Upon a newly booted system, the kernel would start in a state where the new SC scheduling regime is **disabled**. Activation of the new scheduler regime would begin after the first call to change(1). Successive calls with change(1) would have no effect. The scheduling regime would go back to being disabled after the first call to change(0), where successive calls to change(0) would have no effect, but a single call to change(1) would again enable the policy. Notice that when the regime is disabled – **the scheduling should take place as if we never introduced it in the first place (vanilla scheduling)**. See further notes about the regime disabling in the change() system call description.

New Scheduling Regime Ins and Outs:

When the regime is enabled the scheduler will work as follows:

1. Scheduling will take place as normal, unless the **next** scheduled process is SC.
2. When the next process is SC, it will enter the CPU only if it is the **lowest PID** SC process in the system. Otherwise – It will be evicted to the expired queue immediately.
3. When the next process is the lowest PID SC process, it will enter the CPU and may **only** be preempted upon one of the following:
   - A real-time process returned from waiting and is ready to run, or an OTHER process has returned from waiting and has a higher priority.
   - Another CHANGEABLE process entered the run_queue with a **lower PID**.
   - The CHANGEABLE process goes out for waiting.
   - The CHANGEABLE process ends.

Important Notes:

1) **when the currently running task is an SC task and the regime is enabled then the process's time slice must be disregarded, and the process will not evacuate the CPU unless indeed one of the above incidents has occurred.**
2) **In case the next scheduled task is an SC task that should be evicted to the expired queue, the procedure of choosing the next task should be repeated until the scheduler has either chosen a process that is of a different policy, or the process that is SC and has the lowest PID.**
3) **If during the above scheduling procedure all running tasks become expired, you should switch the active and expired queues and repeat the process.**
4) **The sched_yield system call should be disabled for CHANGEABLE processes. In other words, it should return immediately.**

Complexity requirements:

Seeing you are all high-end software developers, you will take complexity into account. You will make sure that checking who is the lowest PID process in the system complies with the following complexity demand: if **n** is the number of CHANGEABLE processes in the system

and L is the number of all the processes combined, then you will make sure that the scheduling procedure does not require more than $O(n)$ in complexity.

All in all, the scheduler must run the processes in the system in the following order

- Real_time (FIFO and RR) processes
- OTHER processes and CHANGEABLE processes (no distinction should be made)
- The idle task

But remember that when the regime is enabled then the above constraints must be preserved.

Forking a CHANGEABLE process

1. The policy of the child is SCHED_CHANGEABLE.
2. The child's static priority is the same as that of its parent.
3. Child's time slice = (ParentTimeSlice div 2)+(ParentTimeSilce mod 2)
4. New parent's time slice = (ParentTimeSlice div 2)

note 1 - 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)
note 2 – the father process must not leave the CPU upon fork() when the policy is enabled to let the child process run.

A Simple Example

Let's assume there are currently only two CHANGEABLE processes running in our system - process A with PID 1700 and priority 101, and process B with PID 1500 ms and priority 139 and the policy is initially disabled.

When the policy is disabled, the scheduler should disregard the PIDs during the scheduling process and therefore, A will typically receive a longer time slice and will be more likely to be chosen to run.

Now let's assume that some other process, denoted C, invokes the ***change***() system call which results in activating the scheduling policy for CHANGEABLE processes.

When C leaves the CPU, another task should take its place. The scheduler would typically choose the process with the highest priority to run. However, in our case, assuming that this process is A (there are no REAL_TIME or OTHER processes with higher priorities available),

it should not be allowed to run, since such behavior violates the principles of the SCHED_CHANGEABLE policy. Therefore, A should be moved to the expired queue, and the scheduler should look for the next available task.

If some other task that is not CHANGEABLE is chosen (has the highest priority among active tasks after A has been moved to the expired queue), it should be given the CPU. However, if the scheduler determines that no such tasks exist, B will be scheduled next.

During the time B has the CPU, even if C is ready to run and has a higher priority, B should not be preempted. The only scenario in which C can reclaim the CPU is if it has gone out for waiting before leaving the CPU and comes back during the time B is in control over the system.

Note that When the policy is enabled, scheduler_tick does not affect CHANGEABLE processes (their time slices remain intact).

Technicalities

New policy

You should define a new scheduling policy SCHED_CHANGEABLE with the value of 3 (in the same place where the existing policies are defined).
Things to note:

- You will notice 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).
- Furthermore, as you have learned in tutorial 4, schedule() disables interrupts and acquires a lock before accessing the run-queue. This ensures that during the time it modifies or reads data from the run-queue, no interrupts can cause the process to leave the data structure in a middle-state and rush to handle those exceptions. Interrupts may in these cases cause the same process to reaccess the run-queue from a different location (function) and remove or add data to it, rendering pointers previously obtained by the same process before handling the interrupt corrupt. In such cases, the process might perform unwarranted memory access when it goes back to schedule() after the interrupt has been handled.

Note that locks have other advantages in SMP systems, but in our case, there is only one CPU, so they are irrelevant to this exercise.

You now realize how critical disabling interrupts when accessing the run-queue is, and therefore, since in some of the system calls you are required to implement for the sake of this assignment you may need to access the run-queue from outside schedule(), you will need to disable the interrupts yourself. Refer to the code of schedule() that is provided to you in tutorial 4 to see how this can be done.

It is highly crucial that you re-enable interrupts before leaving the kernel in all possible scenarios!! If in some case a process leaves the kernel without the interrupts being enabled, the system will crash finally as it will become irresponsive.

Moreover, please avoid unnecessary locking. Locking adds overhead and slows down the system. Also, be careful of attempting to lock an already acquired lock. The lock described in tutorial 4 is known as a spinlock, which forces the process trying to obtain it to spin (busy wait) until the lock is acquired. When the same process acquires the lock and then reattempts to lock again, it will get stuck as the only process that can free the lock is the one that is caught in a loop trying to obtain it. In this case, the kernel will freeze!

The bottom line is: lock only when you must and never double-lock! (In fact, locking is only necessary inside the sys_change() and sys_make_changeable() functions – See description below).

Querying system calls

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

syscall number 243:

```
int is_changeable(pid_t pid)
```

The wrapper will return 1 if the given process is a CHANGEABLE process, or 0 if it is not.

Possible errors:

If no process with the corresponding PID exists - ESRCH

syscall number 244:

```
int make_changeable(pid_t pid)
```

If the caller process and the target process are both not CHANGEABLE, the wrapper should return 0 and the process whose PID is pid should become CHANGEABLE. Note that upon success the target process must immediately be informed whether the regime is enabled or disabled for CHANGEABLE processes, i.e. whether SC processes are currently maintaining FIFO order or not, and act accordingly.

In other words, if there is at least one CHANGEABLE process alive at the time the target process becomes CHANGEABLE, regardless of its state- whether it is running or not, the target process should act the same. If no such processes exist, the neophyte process should consider the regime disabled. It is your responsibility to make sure that at any given time all CHANGEABLE processes conform to the same regime. Possible errors:

If no process with the corresponding PID exists - ESRCH

if the given process or the calling process is a CHANGEABLE process - EINVAL

syscall number 245:

```
int change(int val)
```

The syscall changes the regime for SCHED_CHANGEABLE processes. If val is 0, all CHANGEABLE processes should go back to behaving like regular OTHER processes. If val is 1, all CHANGEABLE processes should start following the regime specified previously.

Important note: as previously mentioned, change() should affect all the CHANGEABLE processes that are currently in the system and all the others that will enter the system in the future. However, at any time, if all CHANGEABLE processes die after the last call to change() and a new process enters the system, it will not be affected and will behave like a regular OTHER process. This must hold until another call to change() is invoked. On success, the return value is 0.

Possible errors:

If val!=1 && val!=0 – EINVAL

syscall number 246:

```
int get_policy(pid_t pid)
```

If the process with PID=pid is CHANGEABLE, the wrapper returns 0 if the policy is disabled and 1 if it is enabled.

Possible errors:

If no process with the corresponding PID exists - ESRCH
If the target process is not CHANGEABLE – EINVAL

Important: All wrapper functions should return -1 on failures and in these cases errno should contain the error value. Moreover, your wrapper functions and service functions must comply with the Linux conventions- The service functions should return the same value as the wrapper functions upon success, and upon failures, the returned value from the service functions should be "–ERROR CODE."
Lastly, the service function's name must match the wrapper's name with "sys_" preceding it.

For any possible error you 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).

For your convenience, we provide you with a template of code that uses the run-queue lock. Feel free to incorporate it into your implementation of the above system calls.

```
rq=this_rq();
spin_lock_irq(rq);
/**
        Your code goes here
**/
spin_unlock_irq(rq);
```

Scheduling

Update the necessary functions to implement the new scheduling algorithm. Note that:

- You must support forking a CHANGEABLE 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 the regime for CHANGEABLE processes is enabled, and a process with a lower PID which is CHANGEABLE as well wakes up, it should be given the CPU immediately. i.e., there shall never be a situation where the policy is enabled, a CHANGEABLE process is running, and there is a more favorable CHANGEABLE process is available. The only exception to this rule is if the process with the lower PID has just woken up and has been activated by the currently running process- which can only happen if the current process has not left the kernel yet after waking up the other process. In such cases *need_resched* **MUST** be set on for the currently running process – We will test this case.

Important Notes and Tips

- 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?
- 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 lower PID processes do 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).
- If something is not defined in the assignment, it will not be tested. You may implement in any way you see fit. Choose an error code that you like, ESRCH will be a reasonable choice.
- The most important note is perhaps that the image of the virtual machine you work on, and that your modification of the kernel code will be tested on is an image of a machine with a **Single** CPU. Therefore, you need not worry about the correctness of the execution in an SMP environment. We are aware that multiple processors running concurrently would require a great deal of time and skills that you might currently lack to meet the demands of the assignment. Therefore, don't attempt to solve the exercise for several concurrent processors and stick to the simpler case. When you need to access the run-queue of your one and only processor, it suffices to use the **this_rq()** macro, which is defined in sched.c.

- 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 miss a file and because of this, the exercise does not work, 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. 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:

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

a. A file named hw2_syscalls.h containing the syscall 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