TECHNION Israel Institute of Technology

Computer Science Department

Operating Systems – 234123

Homework Exercise 2 – Wet

Teaching Assistant in charge:

Mohammed Dabbah

Assignment Subjects & Relevant Course material

Processes, Scheduler.

Recitations 1-4, Lectures 1-3

Assignment Objectives

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-bound processes (תהליכים חישוביים) run **without interruption** and avoid unnecessary context switches to the I/O bound processes (תהליכים אינטראקטיביים).

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 bound processes and will schedule some of the processes running in the system according to a different scheduling algorithm that you will implement and which favors scheduling SCHED_SHORT policies almost uninterrupted by context switches over non real-time processes.

I'm sorry what's SCHED_SHORT policy exactly?

As we have learned in Tutorial 4, each process has its own scheduling policy (like SCHED_FIFO, SCHED_RR, SCHED_OTHER) and Linux scheduling algorithm chooses the next task to run on the CPU based on ready-to-run tasks' priority and their scheduling policy. Our goal is to add a new scheduling policy called SCHED_SHORT to the system and make the scheduling priority as follows (from most important to least important):

- Real time (SCHED_FIFO and SCHED_RR) processes
- SHORT processes
- OTHER processes
- The idle task

Therefore, the new scheduler should ignore SHORT processes as long as exist real-time ready to run processes in the system, and ignore OTHER processes as long as exist ready to run SHORT process.

How do I change the process scheduling policy to the SCHED_SHORT policy?

Only an OTHER process (with SCHED_OTHER policy) might be converted into a SHORT process. This is done by the `sched_setscheduler()` system call (which you have seen in class).

Design Rules:

1. A process may convert itself to the SCHED_SHORT policy.
2. Other processes may convert each other (by supplying the relevant PID to `sched_setscheduler()`)
3. Once a process is SHORT, its **cannot** be converted to any other policy type by the system call.

How it works: The process calls the system call and inserts a **request time**, a number in milliseconds from **1** to **3000**. The process then receives a time slice **equal** to the requested time, and in essence will be scheduled (with "high" priority, as described above) with that single time slice. After that – it will be automatically converted back to an OTHER process. Notice that 3000ms is 10 times the **maximum** allowed time slice in the kernel, as you have learned in class. We conclude that a SHORT process is a quite highly privileged, but this comes at a cost, as you will see in the SHORT penalties section coming up.

With the requested time, you will also supply a new priority value: **`short_prio`**, an integer between 0 and 139.

More details on how to insert this time – in "Policy Parameters".

Scheduling SHORT processes

After all real-time processes have completed running, the CPU should be given to the ready to run SHORT process that has the highest **`short_prio`**.

Just as with OTHER processes (and the **`prio`** value), the priority is sorted from highest (0) to lowest (139). Between SHORT processes with the same priority the order is **FIFO**.

When a new SHORT process enter the system, it will be appended to the end of the queue relevant to its **`short_prio`**. If another SHORT process with a higher priority appears in the run_queue (as a result of converting OTHER process to a SHORT with a higher short priority, or a higher priority SHORT process returned from wait), the higher priority SHORT process should get the CPU as soon as possible (**hint: need_resched**).

To conclude, 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 goes out for waiting.
- The SHORT process ended.
- The SHORT process yields the CPU (with **`sched_yield`**)
- The SHORT process finished its time slice (in which case as we've discussed is converted back as an OTHER process with a **`static_prio`** penalty, detailed below).

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 remaining part. The process will use the remaining time at the next time it is chosen to run.

SHORT Penalties

As you can see if a process knows it has little time to finish, it could gain a lot from converting it's scheduling policy to SCHED_SHORT with a predicted time to its finish, as it would save itself from the cumbersome context switches and be able to get out of the system faster. But there are **disadvantages/penalties** to converting to SCHED_SHORT as well:

1. A SHORT **cannot** use the **`fork`**() or **`clone`**() system call. The return value from the **`do_fork`**() system call in this case should be -EPREM (meaning the calling process doesn't have permission to use this system call). Seeing **`do_fork`**() is shared between **`fork`**() and **`clone`**(), then this effectively constraints both cases.
2. Processes cannot use the **`nice`**() or **`setpriority`**() system calls **on SHORT** processes. In such a case the system call should also return -EPREM. This also means that a SHORT process cannot call these system calls on itself.
3. If the SHORT process did not finish its run during this time slice it gets punished by:
    a. Returning back to be an OTHER process
    b. Its **`static_prio`** will be will be lowered by **7** compared to its original **`static_prio`** before conversion. Due to the score inversion, this actually means: `static_prio=min(old_static_prio + 7,MAX_PRIO-1),` where MAX_PRIO is defined 140.
    c. Its **`sleep_avg`** would be 0.5*MAX_SLEEP_AVG (defined in sched.h), effectively turning its **bonus** to 0.
    d. Finally, its time slice and **`prio`** will be recalculated and it will be inserted back into **`rq->active`**.

To insure the new system isn't used by OTHER process to gain more time slices (by creating child processes, converting them to SHORT processes and then killing them) a SHORT process which calles **`exit`**() before its time slice finishes, the remaining time slice shall not be added to the father's time slice (for that consider modifying **`sched_exit`**() in sched.c) . Notice that this mechanism is something that exists in the kernel code, but we never talked about in class.

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 – sched.h).

Upon changing the policy to SCHED_SHORT using **`sched_setscheduler()`**:

- 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. As we've learned in class, only a superuser (the root user) can change process to be a real-time process. In our new policy we want to allow both superuser and regular user to change OTHER processes to be SHORT processes, but neither a simple user, nor root can change the policy of a SHORT process. In this case you should return EPERM. Note that we will not test against the **init** or **idle** processes.

  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 also encouraged to use any other kernel data structures, macros and methods and modify them as well as reusing existing fields in the **`task_struct`** for your purposes. **In short, do not try to reinvent the wheel**. Seeing this is your implementation, you can do whatever you want, as long as you adhere to the conditions listed above.

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 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 as discussed above.

Querying system calls

You are required to implement the following system calls so we can querry the state of processes in the system:

System Call 243:

```
int is_short(pid_t pid)
```

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

Possible errors:

- If no process with the corresponding PID exists - ESRCH

System Call 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 gets converted back to OTHER.

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

System Call 245:

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

For a 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.

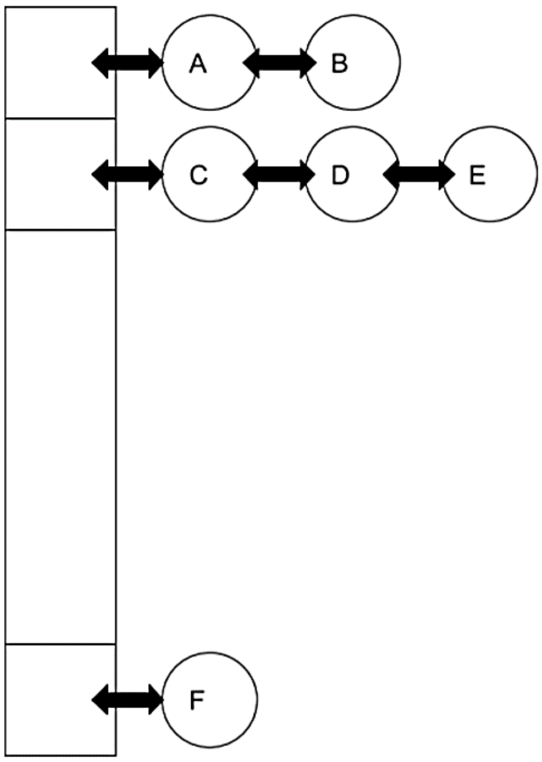
Note that this call does NOT count any real-time processes scheduled before the given process.

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:

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).
- 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. We will of course will not check for this end case.
- In case of an unsuccessful call, system call wrappers should return -1 and update **`errno`** accordingly, like any other system call.
- For **any other** 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).

Important Notes and Tips

- 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.

- All time related values to/from the user are in milliseconds, 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 used by the kernel time slice).

- Reread the recitations on scheduling and context switch (R3-4) and make sure you understand the relationship between the scheduler, its helper functions (**`scheduler_tick`**, **`setscheduler`**, etc.), the run queues, wait queues and context switching.

- 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 kernel 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 encouraged to test as thoroughly as possible.

- Be aware that processes entering 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)

- If something is not defined in the assignment, it will not be tested. You may implement in any way you see fit.

- 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?

- 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).
  Understand how structures such as **`prio_array_t`** work and how you might employ them in your design to ease your workload.

Guidance

You will be provided with the required system calls wrappers, please don't change them.
You may implement your solution in anyway you want as long as you adhere to the above constrains and conditions. With that being said below you can find guidance on which files and methods you ***may*** need to modify (this depends on your implementation).

- **`sched.h (/include/linux/)`**
  - Add the required fields to your implementation.
- **`sched.c (/kernel/)`**
  - **`struct runqueue`** and **`sched_init`**
  - **`rt_task`** macro
  - **`try_to_wake_up`** – You will need to update the logic of when we need to fire a **`need_resched.`**
  - **`effective_prio`**
  - **`scheduler_tick`** - Add/ modify logic of what happens when we discover task's time slice has finished.
  - Modify **`schedule`** – Add the main logic of searching for the next task to schedule.
  - **`set_user_nice, sys_nice`** and **`task_prio`**
  - **`setscheduler`** and **`sys_sched_getparam`**
  - **`schedule_yield`** - SHORT processes act like are like RT processes on yield.
  - Add the required system calls implementation at the end of file.
- **`fork.c (/kernel/)`**
  - Modify **`do_fork`** to prevent SHORT processes from forking.
- **`sys.c (/kernel/)`**
  - Modify **`sys_setpriority`**
- **`entry.s (/arch/i386/kernel)`**
  - Insert support for new system calls

Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of the TAs.
- 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

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form: https://goo.gl/forms/HDFZz3MMtmZxvgXg2

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 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 does not work, you will get a 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 behaves as you expected.

   To open the tar:

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

2) 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
   ```

3) Additional files requirements go here

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain only the X files, without directories.

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

```
zip final.zip kernel.tar.gz submitters.txt other files
```

The zip should look as follows:

```
zipfile -+
         |
         +- kernel.tar.gz
         |
         +- submitters.txt
         |
         +- other files
```

Important Note: when you submit, **retain your confirmation code and a copy of the file(s)**, in case of technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Have a Successful Journey,

The course staff