Homework 2 Wet

Due Date: 17/5/18, 23:30

Teaching assistant in charge:
- Yehonatan Buchnik

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-wet , put them in the hw2-wet folder

Only Idan, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly.

Before starting, make sure your kernel is clean from the changes of HW1

Introduction

In the last weeks you have seen a various of scheduling algorithms which are generally partitioned into two classes: Batch scheduling and Preemptive scheduling. A common feature of those algorithms is the attempt to optimize the ***response time*** of the scheduled jobs.
In this assignment you will learn and develop a different type of scheduling policy that tries to optimize the ***unfairness*** by using the ***proportional share*** paradigm. Combined with the Dry assignment, you will analyze the new scheduling policy and gain a profound knowledge in the scheduling paradigms and in the 2.6 linux scheduler in particular.

Proportional sharing

In the proportional sharing paradigm every process receives a certain percentage of the CPU time that is proportional to its share in the system. In this paradigm a process may start later (compare to Round Robin for example), but each process receives a fair portion of the CPU time.
Notice that the 2.6 scheduler is trying to take that fairness in count as each process receives in every epoch a time slice that is proportional to its static priority. Alas, the scheduler implementation is highly complicated and it doesn't offer a full ***proportional share*** method.

Lottery scheduler

So, how can we design a simple and intuitive proportional scheduler?
Lucky for us, this question was already discussed in the late 80'. The solution is the ***Lottery scheduler.***
In Lottery scheduler, each process holds a number of lottery tickets. On every context switch the scheduler is randomly choosing a number. The process that holds the ticket with the chosen number wins the lottery and starts running. If the cards' number of each process is proportional to the process's share in the system, the randomly lottery optimize (in expectation) the CPU ***unfairness***.
Yet, we still have to understand how to share the tickets fairly. In our implementation we will rely on the priority mechanism that already exists in linux. The higher priority a process has, the higher its share becomes. For example, considering two processes A,B such that A's priority is 1 and B's priority is 139, then the sum of the tickets in the system is 140. A holds 139 tickets and B holds only 1 ticket.
For a more comprehensive explanation please visit here.
The implementation is built out of two parts. First you will implement a scheduler logger in order to track the scheduling course. Then you will implement the actual Lottery scheduler.

Part 1 – Building a scheduler logger

In this part you will implement the logger infrastructure. Later, more fields will be added, hence, we advice you to build a flexible logger such that adding more logged information can be done easily.
For now a logger record contains the following data:

```
typedef struct {
   pid_t prev;
   pid_t next;
   int prev_priority;
   int next_prioroty;
   int prev_policy;
   int next_policy;
   long switch_time;
} cs_log;
```

`pid_t prev`	The previous process's PID
`pid_t next`	The next process's PID
`int prev_priority`	The previous process's priority (`prev->prio`)
`int next_prioroty`	The next process's priority (`next>prio`)
`int prev_policy`	The previous process scheduling policy (`prev->policy`)
`int next_policy`	The next process scheduling policy (`prev->policy`)
`long switch_time`	The time in which the context switch happened (in jiffies)

Below are listed the system calls that implement the logger's logic:

syscall number 243:

```
int enable_logging(int size)
```

After invoking the `enable_logging`, up to ***size*** records are able to be logged by the logger.
Return values:

- On success return 0.
- On failure

1. The logger is already enabled 2. size < 0	return -1 errno should contain EINVAL
1. Any memory failure	return -1 errno should contain ENOMEM

syscall number 244:

```
int disable_logging()
```

This system call disabling the logger. After invoking `disable_logging` no record is supposed to be logged from then on. Note that this system call doesn't empty the logger but only disables it.

Return values:
- On success return 0.
- On failure

1. The logger is already disabled	return -1 errno should contain EINVAL
1. Any memory failure	return -1 errno should contain ENOMEM

syscall number 245:

```
int get_logger_records(cs_log* user_mem)
```

This system call copies the entire log data into *`user_mem`* and empties the log.

Parameters:
- *`user_mem`* - Pointer to the user memory where the log is asked to be copied to. *`user_mem`* is allocated by the user. On success *`user_mem`* will contain all the log's records.

Return values:
- On success return the number of records that have been copied.
- On failure

1. user_mem is NULL *2. any memory failure*	return -1 errno should contain ENOMEM

Part 2 – The Lottery scheduler

In this part you will implement the Lottery scheduler. The scheduler is randomly and hence it can yield different results in each run, but in expectation, the CPU time of a process should be proportional to its share in the system.

Detailed Description

The system

- **Number of tickets in the system:** The number of tickets in the system can be defined (denoted as NT) in two ways:
    - The sum of tickets of all the runnable processes (see below)
    - A special system call that sets the number of tickets in the system. In such a case, the number of tickets may be lower than the sum of the tickets of the runnable processes (but never higher). If so, only the processes with the highest priorities may participate in the lottery.
      For example, in a system with the following specification:
        i. $NT = 10$
        ii. There are only four runnable processes:
            1. $p_1 -> prio = 135$ and p_1 holds 5 tickets.
            2. $p_2 -> prio = 136$ and p_2 holds 4 tickets.
            3. $p_3 -> prio = 137$ and p_3 holds 3 tickets.
            4. $p_4 -> prio = 138$ and p_1 holds 2 tickets.

      only p_1, p_2 and p_3 can participate in the lottery such that p_1 has 5 tickets, p_2 has 4 tickets and p_3 has only 1 ticket.
      If, p_1, p_2, p_3 and p_4 are all with the same priority, then the share of the tickets is with respect to their order in the queue. Meaning that there might be a various of correct shares, depending on the specific run of the system.
- **Choosing the next process to run:** In the original 2.6 linux scheduler the next process to run is the process with highest priority. If that process is a real time process it may grab the CPU until it finishes. In the Lottery scheduler, even in the presence of real time processes, any runnable process may be the next to run, depends on the lottery result (but any other functionality of processes with `prio < 100` remains the same). On every context switch the scheduler randomly chooses a number $i \in [0, NT-1]$, the process that holds the i_{th} ticket is the next process to run. Obviously, this can be done in $O(n)$ steps for every context switch (how?). But, we would like to achieve a better complexity;
  Let $k = max\{number\ of\ runnable\ processes\ in\ priority\ x \mid x \in [1, 139]\}$, then choosing the next process should be done in at most $O(1) + k$ steps for each context switch.
- **Scheduling policy:** When the Lottery scheduler is on, all the processes in the system are being under the `SCHED_LOTTERY` policy. When the time slice of a

process has finished the process returns to the end of the appropriate queue (according to `prio`) in `active` and its `time_slice = MAX_TIMESLICE`. Notice that the result is that there are no epochs under the `SCHED_LOTTERY` policy. Changing the policy (when the Lottery scheduler is on) by the appropriate linux system calls (such as `sched_setscheduler`) is forbidden. Changing the policy back to its original policy (`SCHED_OTHER, SCHED_FIFO` and `SCHED_RR`) is done only via a special new system calls (see below). Similarly, changing the policy to `SCHED_LOTTERY` when the Lottery scheduler is off, is forbidden. In these cases the policy changing should fail with some negative return value (not specified).

- **Other functionality:** Any other linux kernel functionality should stay the same. Namely, processes may sleep, wait, or yield the CPU, interrupts should regularly fired off, the priority of a process may changed and so on.
  Note that processes with `rt_priority` $\in [0, 99]$ should act as their original functionality except that there might be a process with higher `prio` that can run before them - depends on the lottery result.

The process

- **Switching to Lottery scheduling:** When the Lottery scheduler is turned on each process should be changed as following:

`p->time_slice`	set to `MAX_TIMESLICE`
`p->policy`	set to `SCHED_LOTTERY`

- **The process's tickets:** As mentioned earlier, each process holds tickets that represent its share in the system and allows it to win the lottery. The number of tickets that a process p holds is proportional to its priority according to the following formula:

$$\textit{p's number of tickets} = \texttt{MAX\_PRIO} - \texttt{p->prio}$$

  Using this formula gives a process with high priority a better chance to win the lottery. Note that as the effective priority might changed the process's tickets' number might also be changed respectively.
- **Switching back to the original scheduler:** Switching the scheduler to the original one requires each process to restore its previous values of the above fields. Note that `p->time_slice` is needed to be recalculated according to the process's priority.

The logger

Add to your `cs_log` struct another field named as `int n_tickets`. This field hold the number of tickets in the system and is required to be valid only when the Lottery scheduler is on. Add this field in the end of the `cs_log` struct, such that it is the last field of `cs_log`.

Implementation

Below the system calls that implement the scheduler logic are listed:

syscall number 246:

```
int start_lottery_scheduler()
```

This system call starts the Lottery scheduler. After invoking `start_lottery_scheduler` the processes will be scheduled only by the lottery scheduling policy (and will be changed as mentioned earlier).
Note that while handling and the runqueues you must disable the interrupts in the system.

Return values:

- On success return 0.
- On failure

1. The Lottery scheduler is already on	return -1 errno should contain EINVAL

syscall number 247:

```
int start_orig_scheduler()
```

This system call restores the original 2.6 linux scheduler and starts new epoch. After invoking `start_orig_scheduler` the processes will be scheduled only by the original 2.6 linux scheduling policies. When restoring the original scheduler a process is setting back to run with the policy he had when `start_lottery_scheduler` has been invoked.
In a case of success, a new epoch of the 2.6 linux scheduler starts.
Note that while handling and the runqueues you must disable the interrupts in the system.

Return values:

- On success return 0.
- On failure

1. The original 2.6 linux scheduler is already on	return -1 errno should contain EINVAL

syscall number 248:

```
void set_max_tickets(int max_tickets)
```

This system call sets the max available tickets in the system. Invoking `set_max_tickets` sets $NT = min\{\sum_{p\ is\ runnable} p's\ number\ of\ tickets,\ max\_tickets\}$.

Note that as $\sum_{p\ is\ runnable} p's\ number\ of\ tickets$ may change on each context switch, hence, NT should be recalculated on every context switch.

If `max_tickets` ≤ 0 the limit should be canceled.
This system call never fails.

Description of your solution

Attach to your submission a description of your solution (up to one page) in PDF format.

Important Notes and Tips

- Reread the tutorial on scheduling and make sure you understand the relationship between the scheduler, its auxiliary functions (scheduler_tick, setscheduler, etc.), the run_queue, wait queues and context switching.
- Before starting, try to read the kernel's code and understand exactly its flow.
- 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 scheduling algorithm?
- In order to generate random numbers in the kernel, include `linux/random.h` and use the `get_random_bytes` method.
- Note that allocating memory (`kmalloc` and `kfree`) from the scheduler code is dangerous, because `kmalloc` may sleep.
- 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.
- You are well versed in the construction and usage of system-calls. Use them! You are more than welcome to create system-calls for your personal testing-purposes, and are encouraged to use them to incrementally build your solution.
- 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 be wary of them.
- 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
- Files you should consider changing in this exercise include (but are not limited to):
    - sched.c (in the kernel folder)
    - sched.h (in include/linux/)
    - entry.s (in arch/i386/kernel)

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:

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

a. Additional files requirements go here

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
hw2_description.pdf
```

The zip should look as follows:

```
zipfile -+
        |
        +- kernel.tar.gz
        |
        +- submitters.txt
        |
        +- hw2_syscalls.h
        |
        +- hw2_description.pdf
```

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