Homework Wet 2

Due date: 19/12/2012 12:30

Teaching assistant in charge:

- Anastasia Braginsky

E-mails regarding this exercise should be sent only to the following email: cs234120@cs.technion.ac.il with the subject line: cs234123hw2_wet.

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 CPU bounded processes, to let them run in **Bounded Round Robin** with **Mutable Quantum** (MQ), 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_MQ**, is designed to support important 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 **MQ-process**.

Only an OTHER-process (with SCHED-OTHER policy) might be converted into a MQ-process. This is done by the `sched_setscheduler()` system call. When the policy of a process is set to SCHED_MQ, the caller of the `sched_setscheduler()` should also inform the operating system of the **number of trials** of the process. The number of trials must be an integer in the set of {1, 2, 3, … 100}. The MQ-process will get a time-slice longer than OTHER-process's time-slice, but MQ-process could try to finish its run in 1, 2, … or 100 trials, each in the length of its time-slice (that is different for each trial). The MQ-process's time-slice is calculated at the beginning of each trial *n* according to the following function:

$$(1+1/n)*\text{TASK\_TIMESLICE}(p)$$

Pay attention that the first trial is done with the initial time-slice inherited from when the process was still OTHER-process. So *n* actually starts from 2. How the MQ-child-process's initial time-slice is calculated will be explained later.

A MQ-process that used all of its trials and didn't finish will be considered an **Overdue-MQ-process**. A MQ-process or Overdue-MQ-process **can never be** changed back into an OTHER-process (or a real-time process). Once a process has become MQ, it will remain MQ (might be Overdue-MQ), until it exits.

You as the kernel designers may decide to maintain any other kernel data-fields needed for a MQ-process.

Scheduling policies order

Any MQ-process that is not overdue, will receive higher priority than the OTHER-processes. However, a MQ-process that is overdue, 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. MQ-Processes
3. OTHER processes
4. Overdue-MQ Processes
5. The idle task

Therefore, the new scheduler should ignore MQ-processes as long as there are real-time ready-to-run processes in the system, and ignore Overdue-MQ-processes while there are any OTHER ready-to-run processes (also in expired (!) priority queue). In general, SCHED_OTHER and SCHED_MQ scheduling policies are different and not related policies. For example, MQ-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 MQ-processes, and only at the end, when you are convinced that it works properly, change it and give the MQ-processes the higher priority. So, while developing, if a MQ-process is running and an OTHER-process wakes up, the MQ-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 MQ-process doesn't give up the processor for a regular process.
The reason for this is that if you run the system with the MQ-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 MQ-processes

The CPU should be given to the ready-to-run MQ-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 MQ-processes. After that, P runs in

Round-Robin (RR) with other MQ-processes having same priority. P can participate in RR only its *"number of trials"* times.

A MQ-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-MQ-process (and choose a different process to run accordingly).

As stated above, between MQ-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) MQ-process that has same priority, till the end of P's time-slice. However, if another MQ-process with higher priority appears in the run_queue, the higher-priority MQ-process should get the CPU. The lower-priority MQ-process that left the CPU should remember the remained part of its time-slice and use it later – counting all the time-slice usage as a single trial.

Thus, a MQ-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 MQ-process returned from waiting, and it has higher priority.
3. The MQ-process forked, and created a child (see explanation in the next section).
4. The MQ-process goes out for waiting.
5. The MQ-process ended.
6. The MQ-process finished this time-slice (it may return to another trial)
7. The MQ-process finished this time-slice and it was its last trial (it has more code to do). The process became an overdue process.
8. The nice() call has changed the priority of some lower priority MQ-process to have higher priority

Note that a MQ-process is not allowed to yield the CPU! An attempt to yield should result in an error (return -1). Also note that if there is only one MQ-process for some priority, it simply gets "number of trials" time-slices with no interruptions.

Scheduling Example

In the table below you can see a part of a scheduling of three MQ-processes. The events are explained. The scheduling doesn't end at the last column, it continues further according to the rules. For timestamp calculations we used approximate values and n started from 2 as explained.

Processes with static priority (s. pr.)	**Time-slices & number of trials**	**The scheduling of the processes**					
P_1 with s. pr. 120	151, 226, 201 milli-seconds & 3		CPU goes to P_1 (P_2 could also be chosen). After 151 ms, CPU is taken from P_1, since P_1's time-slice is used.				P_1 gets one more time-slice since it has one more trial. P_1 runs for more 226 ms and finishes its second time-slice, but it has third trial. P_1 uses 10 ms more and finishes its code.
P_2 with s. pr. 120	151, 226, 201 milli-seconds & 2			P_2 runs for 10 ms, then CPU is taken from P_2, because P_3 returns from waiting.		P_2 gets the CPU, but finishes after 10 ms. P_2 is done.	
P_3 with s. pr. 110	225, milli-seconds & 1	P_3 runs first, due to the higher priority. After 100 ms, P_3 goes to wait.			P_3 runs for more 125 ms, but didn't finish. The CPU is taken from P_3, since its time-slice ended and it had only 1 trial. P_3 turns to be overdue.		

Time

Forking a MQ-process

i. The policy of the child is SCHED_MQ.
ii. The parent gives up the CPU and the scheduler goes to the next task according to the SCHED_MQ scheduler.
iii. The child's static priority is the same as of its parent.
iv. The child's number of trials is the same as of its parent at the fork time.
v. The child's initial time-slice (for its first trial) has the same value as of its parent at the fork time. Later it is calculated for each trial as explained above.

Scheduling Overdue-MQ-processes

Overdue-MQ-processes do not consider their priority, as if they all have the same priority. We can imagine a queue of ready-to-run Overdue-MQ-processes waiting for CPU. Among the Overdue-MQ-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-MQ-process should run until it finishes or goes to wait. Of course any other scheduling has a higher priority than Overdue-MQ.

For summary, an Overdue-MQ-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.
3. The process ended.

Similar to a MQ-process, am Overdue-MQ-process cannot yield the CPU.

Forking an Overdue-MQ-process

When a Overdue-MQ-process is forking, the child is also Overdue-MQ and it enters the Overdue-MQ 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.
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 MQ-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_MQ` with the value of 4 (in the same place where the existing policies are defined).

Upon changing the policy to `SCHED_MQ` 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 MQ-process, "Operation not permitted" error should return.
  A user should also fail to change the policy of a process of another user to SCHED_MQ.
- 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 MQ-processes. But, again, remember that a MQ-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_MQ algorithm must extend this struct to contain other parameter of the algorithm.

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

When a process is turning to be MQ, initialize `sched_param->sched_priority` to zero. Anyway while `sched_setscheduler()` is invoked for SCHED_MQ it should not change the process priority.

Notice that process cannot become Overdue-MQ via the above system calls, a MQ-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 MQ-process to a different value than initial.

Querying system call

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

- syscall number 243:
  ```
  int is_MQ(int pid)
  ```

The wrapper will return 1 if the given process is a MQ-process, or 0 if it is already overdue.

In case of an unsuccessful call (process is not a MQ- or Overdue-MQ-process) wrapper should return -1 and update errno accordingly, like any other syscall. In such a case, 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 MQ- and Overdue-MQ-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 MQ-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 (not possible for MQ/Overdue-MQ processes).
4. A MQ-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 MQ, 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 244. 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_MQ.

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 i^{th} 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?

4. Important Notes and Tips

- 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 MQ-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.

Do not print the electronically-submitted source code.

Handwritten assignments will not be accepted.

Have a Successful Journey,
The course staff