Homework 1 Wet

Due Date: 16/11/17 23:30

Teaching assistant in charge:
- Yehonatan Buchnik
- 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 hw1, put them in the hw1 folder

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

Your goal in this assignment will be to add new system calls to the kernel’s interface, and to change some existing system calls. While doing so you will gain extra knowledge in compiling the kernel. Furthermore in this exercise, we will use VMware to simulate a virtual machine on which we will compile and run our "modified" Linux. You will submit only changed source files of the Linux kernel.

**General Description**

We would like to add some new logging services to the linux kernel. The services should record part of the process activities and the user can get those records through system calls.

**Detailed Description**

You need to implement code wrappers and the corresponding system calls. For example: `enable_syscalls_logging` is a code wrapper and `sys_enable_syscalls_logging` is a system call (see the slides for tutorial 2).

We would like to record for any process it’s system calls invocations. Let’s define the following struct:

```c
struct syscall_log_info {
    int syscall_num;
    int syscall_res;
    int time;
};
```

Where `syscall_num` is the control number of the syscall which is currently invoking (remember: each system call has an associated number passed to `system_call()` as a parameter).,

`syscall_res` is the returning value of that syscall and `time` is the clock ticks (in jiffies) in which the syscall returned.

For example: by invoking the getpid (it’s designated number is 20) system call by the process with PID=10 after 42 clock ticks, we would like to record: {20, 10, 42}

After enabling the logging for a specific process we would like to record **any and all system calls which that process invokes. The logging process must apply to every system call that exists within the system, AND any system call that might be theoretically implemented in the future.**

**Meaning:** After invoking `enable_syscall_logging` (see below) every syscall invocation will be recorded in the mentioned format.
Hint: While implementing, consider that the main mechanism of the system calls goes through the system_call assembly routine (located in arch/i386/kernel/entry.S). In order to catch every system call invocation you may want to modify this routine in order to perform the logging operations before and/or after the system call invocation.

Note:
1. For the sake of the home assignment, please ignore handling of processes containing multiple threads. That is, assume tgid=pid for all processes, and ignore memory sharing (or other security) issues.
2. Accessing the clock is through a global Kernel variable called jiffies

Code Wrappers

int enable_syscalls_logging(pid_t pid, int size)

Description
Enable the logging of system calls by the process with with PID=pid. The log can contain up to size records.

Return values
- On success: return 0.
- On failure: return -1
  - If pid < 0 errno should contains ESRCH
  - If no such process exists errno should contains ESRCH
  - If SIZE < 0, errno should contains EINVAL
  - If enable_syscalls_logging already invoked and disable_syscalls_logging isn’t set, errno should contains EINVAL
  - On memory allocation failure errno should contains ENOMEM
  - On any other failure errno should contains EINVAL

int disable_syscalls_logging(pid_t pid)

Description
Disable the system-call logging by the process with PID=pid and release the log resources. After invoking disable_event_logging any future system calls shouldn’t be logged anymore. This function also nullifies the state of system-call logging, meaning, after invoking it we can invoke the enable_syscalls_logging function again without error, starting the logging procedure again from scratch.

Return values
- On success: return 0
• On failure: return -1
  • If pid < 0 errno should contains ESRCH
  • If no such process exist errno should contains ESRCH
  • If enable_syscalls_logging haven't invoked yet errno should contains EINVAL
  • On any other failure errno should contains EINVAL

int get_syscalls_log(pid_t pid, int size, syscall_log_info* user_mem)

Description
Return the firsts \( size_{th} \) records of the the syscalls log of the process with PID=\( pid \) in user_mem (which is allocated by the user).

Note that this function should also empty the first \( size_{th} \) records from the log.
Note that this function enables a given process to access the system calls log of itself, or an entirely different process, depending on the given PID.

Return value
• On success: return 0
• On failure: return -1
  • If pid < 0 errno should contains ESRCH
  • If no such process exist errno should contains ESRCH
  • If size > log_size, errno should contains EINVAL
  • If size < 0, errno should contains EINVAL
  • If the logging feature isn't set errno should contains EINVAL
  • On memory allocation failure errno should contains ENOMEM
  • If less than size log records were were passed to the user, errno should contains ENOMEM (see the man of copy_to_user for more details)
  • On any other failure errno should contains EINVAL

Clarifications
1. The log is cyclic, meaning: if the log is full you should continue record above the oldest records.
2. Be wary of memory leaks
3. In this exercise you need to write a little assembly code snippet, but the main logic of the recording operations should be written in C, therefore be aware of the C Calling Conventions while invoking C function from an assembly code.
4. You may want to have a look on the kmalloc, kfree and copy_to_user functions.
5. After forking, the logging system in the son should be nullified even if the father process is currently logging system calls - meaning the son has an empty log and is NOT recording system calls. (Remember that fork() itself is also a system call!)
6. You require to log a system call after it's returning, the obvious result is that the disable_syscalls_logging is always exempt from the logging.
7. When handling an assembly code, read carefully the full specification of the op-codes (it can be found on the web).

**Code Wrapper**

As was mentioned earlier, you need to implement a code wrapper for your system calls.

Below is an example of the code wrapper for my_system_call (#244). Follow this example to write the wrappers.

```c
int my_system_call (int p1, char *p2, int p3) {
    unsigned int res;
    __asm__ {
        "int $0x80;"
        : "=a" (res)
        : "0" (244), "b" (p1), "c" (p2), "d" (p3)
        : "memory"
    };
    if (res >= (unsigned long)(-125))
    {
        errno = -res;
        res = -1;
    }
    return (int) res;
}
```

**Explanation of inline assembler:**

The assembler structure is:

```c
asm ( assembler template
    : output operands   (optional)
    : input operands    (optional)
    : clobber list      (optional)
);
```

The asm is volatile to tell the compiler it has side effect besides the output operands. In our case it means that even if res is never used the compiler may not delete this assembly block (compilers sometimes do that for optimization).

The only command we need to issue in assembly is "int $0x80".
The rest of the preparation is done by the compiler assuming we describe the operand correctly.

The operands are numbered according to the order that they specified and are described below
Output operands:
%0: "=a" (res) - the "=" means it output and "a" means it should be in the register eax, the (res) say that it should be put in the variable res.

Input operands:
%1: "0" (244) - the "0" say we want this operand to have the same constraints as operand %0, which in our case mean to be in eax. The (244) says we want it to have the value 244 when the assembly block begins. Which should be the corresponding system call's ID number (This is the value  system_call() will use to find our declared system call in the table defined in entry.s)

%2: ",b" (p1) – "b" means we want this operand to be in ebx and (p1) means we want it to have the value of p1
%3: ",c" (p2) – "c" means we want this operand to be in ecx and (p2) means we want it to have the value of p2
%4: ",d" (p3) – "d" means we want this operand to be in edx and (p3) means we want it to have the value of p3

clobber list:
"memory" tells the compiler that the asm block may write to memory that wasn't specified as an output operand.

Useful links:
http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

What should you do?

Use VMware, like you learned in the preliminary assignment, in order to make the following changes in the Linux kernel:

1. We highly recommend you to go through the First Steps in Linux Kernel Development - A Walkthrough under the additional materials in the webcourse. This may save you a lot of time!
2. Put the implementation of the new system calls in the file kernel/hw1_syscalls.c and arch/i386/kernel/hw1_logging.c that you will have to create and add to the kernel. Update the makefiles in those directories to compile your new file too. (Tip: add it to obj-y).
3. Figure out where is the central point in which every system call invocation must go through, and understand how to add your logging logic there.
4. Update entry.S (add system call numbers and references in the syscall table).
5. Make any necessary changes in the kernel code so the new system calls can be used like any other existing Linux system call. Your changes can include modifying
any .c, .h
or .S (assembly) file that you find necessary.

6. Make necessary changes in file fork.c and sched.h

7. Update more files if needed.

8. Recompile and run the new kernel like you did in the preliminary assignment.

9. Put the wrappers functions in hw1_syscalls.h, note that hw1_syscalls.h is not part of the kernel, the user should include it when using your system calls This also means you don’t need to recompile the entire kernel when modifying the header.

10. Boot with your new Linux, and try to compile and run the test program to make sure the new system calls work as expected.

11. Submit kernel.tar.gz, submitters.txt and hw1_syscalls.h (see below)

Did it all? Good work, Submit your assignment.

Important Notes and Tips

- First, try to understand exactly what your goal is.
- Figure out which data structures will serve you in the easiest and simplest way
- Figure out which new states you have to save and add them to the task_struct (defined in sched.h).
- Figure out in which exact source files you need to place your code and where exactly in each file.
- Do not reinvent the wheel, try to change only what you really understand, and those are probably things related to the subjects you have seen in the tutorials.
- Debugging the kernel is not a simple task, use printk to print messages from within the kernel. Whenever possible - write and compile short and simple segments of code and make sure they work before expanding on them (For example, writing a preliminary system call that simply returns a number, and making sure that works, before adding functionality to it)
- The linux developers wrote comments in the code, read them, they might help you understand what's happening.
- You are not allowed to use syscall functions to implement code wrappers, or to write the code wrappers for your system calls using the macro _syscall1. You should write the code wrappers according to the example of the code wrapper given above.
- If there is more than one error when calling a syscall you should return the first one by the order in the function description.
- Write your own tests. We will check your assignment with our test program
- We are going to check for kernel oops (errors that don't prevent the kernel from continue running such as NULL dereference in syscall implementation). You should not have any.
   If there was kernel oops, you can see it in dmesg (dmesg it's the command that prints the kernel messages, e.g. printk, to the screen).
   To read it more conveniently use: dmesg | less -S
- Linux is case-sensitive. entry.S means entry.S, not Entry.s, Entry.S or entry.s.
- You can assume that the system is with a single CPU.
You should use kmalloc and kfree in the kernel in order to allocate and release memory. If kmalloc fails you should return ENOMEM. For the kmalloc function use flag GFP_KERNEL for the memory for kernel use.

Pay attention that the process descriptor size is limited. Do not add to many new fields. Also, add your fields at the end of the struct because the kernel sometimes uses the offsets of the fields.

Start working on the assignment as soon as possible. The deadline is final, NO postponements will be given, and a high load on the VMWare machines will not be accepted as an excuse for late submissions.

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 -cf 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.

Furthermore - make sure upon submission that you receive the submission number and record it! It could save you a lot of pain.

To open the tar:

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

**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 hw1_syscalls.h
```

The zip should look as follows:

```
zipfile +- 
 | +- kernel.tar.gz 
 | | +- submitters.txt 
 | | +- hw1_syscalls.h
```

*Have a Successful Journey,*  
The course staff