TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems – 234123

Homework Exercise 1 – Wet

Teaching Assistant in charge:

Yehonatan Buchnik

Assignment Subjects & Relevant Course material

Processes, System Calls, Calling Conventions

Recitations 1-2 & Lecture 1, HW0

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.
Please read the whole document before you start working.

General Description

We want you to dynamically prohibit certain processes from calling some system calls. To do so, you will add new features to the Linux kernel wherein every process is associated with a specific restriction level and a restrictions list. Then, you will modify the *system_call* routine, that you have learned about, such that a process will be able to invoke only the system calls that are not restricted by its restrictions list and its restriction level. Finally, you will create a logging service that records forbidden activities of processes in the OS.

The policy is defined as follows:

- **Turning On The Feature**: By default, all processes will not be affected by the new feature.
    - You will define a system call which gets the *PID* of a process, a *restriction level* and a *restrictions list* of $[syscall\ number,\ syscall\ restriction\ threshold]$ and turns the feature on for the process with the given PID with respect to the received restriction level and restrictions list.
    - Every process may activate the feature for any other process (including itself).
      You can assume that we will not test your code for essential system processes (such as *idle or "swapper"* (PID = 0) and *init* (PID = 1)).
- **Restriction Level/threshold**: Every process which the feature has been invoked for it, is assigned with *restiction_level* in the range of [0, 2]. Similarly, any system call in the given restrictions list is associated with a *restriction_threshold* in the range of [0,2].
- **Effects of the restriction level and restrictions list**:
    - For each system call in the process restrictions list, if the process has a *restriction_level* which is strictly lower than the *restriction_threshold* of that system call, then the call should fail and return -1. We define such behavior as a **forbidden activity**.
    - You should not restrict any other system call.

- For example, (See the system calls specifications below):

```
int main() {
   pid_t pid = getpid(); // getpid system call number is 20
   scr r = {20, 2};
   sc_restrict(pid, 0, &r, 1);
   getpid(); // should fail (return -1), errno == ENOSYS
   set_proc_restriction(pid, 2);
   getpid(); // should succeed
   return 0;
}
```

- **Recording Forbidden Activities**: You must define a separate log for each process. If a process attempts to perform a forbidden activity, then this activity should be recorded in the log of that process. Specific details on the implementation of the log are discussed below.

Important Notes:

- If a process has invoked a restricted system call, the system call should fail and return -1 . In addition, errno should contain ENOSYS.
- Note that this new feature should also work for any future system call we may add to the OS .
- You are not required to handle any failure that might occur due to incorrect using in the feature (for example, restricting a process from dying).
- You may assume that we will not restrict system processes (init, swapper ..).
- Advanced Note: for the sake of the home assignment, please ignore handling processes containing multiple threads. That is, assume tgid=pid for all processes, and ignore memory sharing issues.
- You may assume a single processor\core system.

Detailed Description

Restrictions list

A restricted process has a restrictions list which consist of system call numbers and their desired restriction threshold.
For this purpose, we define the following struct that represents an element in the restrictions list:

```
typedef struct sys_call_restriction {
      int syscall_num;
      int restriction_threshold;
} scr;
```

- **syscall_num -** stores the system call number for which we would like to set the restriction
- **restriction_threshold -** stores the restriction threshold of the system call.

Note that the struct should be of type *scr* (using typedef). Defining it differently may cause your code not to pass our tests.

The log records' description

We would like to record the forbidden activities of a given process.
For this purpose, we define the following struct that represents a forbidden activity log record:

```
typedef struct forbidden_activity_info {
      int syscall_num;
      int syscall_restriction_threshold;
      int proc_restriction_level;
      int time;
} fai;
```

- **syscall_num -** stores the number of the involved system call.
- **syscall_restriction_threshold -** stores the restriction threshold of the involved system call.
- **proc_restriction_level** - stores the involved process restriction level **at the time** the forbidden activity occurred. Note that the process's restriction level might be changed through time via ad-hoc system calls, as we will discuss below.
- **time** - stores the clock ticks (the value in jiffies) at which the forbidden activity occurred
  (The clock ticks can be retrieved from a kernel global variable named *jiffies*).

Note that the struct should be of type `fai` (using typedef). Defining it differently may cause your code not to pass our tests.

Important Notes:

- You should define these structs in both kernel and user code (hw1_syscalls.h)
- After logging forbidden activities, even when all the restrictions have been removed, the log should still be available for reading.
- The log is cyclic and has a predefined size (of **100** records). A record is removed from the log only if it is the oldest record, the log is full and a new record should be inserted.
- Once it created, the log will be deleted only on a process totally termination (it means that one should be able to read the log of a zombie process).

Code wrappers

You are required to implement the following code wrappers and the corresponding system calls. For example: *sc_restrict* is a code wrapper and *sys_sc_restrict* is a system call (see Recitation 1, from slide 36 onwards).

System call number 243

```
int sc_restrict (pid_t pid ,int proc_restriction_level, scr* restrictions_list, int list_size)
```

Description

Imposes the restrictions for the process with PID=*pid* with respect to the given *proc_restriciton_level* and the *restricitions_list*. The restriction level of this process should be set to *proc_restriction_level*. In addition, from this point on, the log should record (at most) the last **100** forbidden activities that this process has performed.

- If this method was called more than once than its last invocation takes effect.
- The restrictions list may be empty. In this case, de facto, the process has no restrictions.
- Use the `copy_from_user` method in order to copy the content of *restrictions_list* from the user memory to the kernel memory.

Parameters

pid	The process for whom we would like to set the restrictions
proc_restriction_level	The process restriction level. From now on, every invocation of a system call which is configured in *restrictions_list* that has a strictly greater *restriction_theshold* than *proc_restriction_level*

	should fail and will be recorded as a forbidden activity.
restrictions_list	A configuration list in which every item contains a system call number and its desired *restriction_theshold*. Every system call that is not in this list should not be affected. **You may assume that the data within *restrictions_list* is valid**.
list_size	The number of elements in *restricions_list*. You may assume that this number always match the actual list size.

Return values

- On success: return 0.
- On failure: return -1
    - If pid<0 errno should contain *ESRCH*
    - If no such process exists errno should contain *ESRCH*
    - If proc_restriction_level<0 or proc_restriction_level>2 errno should contain *EINVAL*
    - If size<0 errno should contain *EINVAL*
    - On memory allocation failure errno should contain *ENOMEM*
    - On memory copy failure errno should contain *ENOMEM*
    - On any other failure errno should contain *EINVAL*
- If there is more than one error you should return the first one by the above order.

System call number 244

int set_proc_restriction (pid_t pid ,int proc_restriction_level)

Description

Sets a new restriction level for the process with PID=*pid*. This method only sets a new restriction level regardless of the process's restrictions list.

- If this method was called in parallel to the *sc_restrict* method, then the one that was performed last should takes effect.
- If this method was called more than once than its last invocation takes effect.

Parameters

pid	The process for whom we would like to set the restrictions
proc_restriction_level	The process new restriction level

Return values

- On success: return 0.
- On failure: return -1

    - If pid<0 errno should contain *ESRCH*
    - If no such process exists errno should contain *ESRCH*
    - If proc_restriction_level<0 or proc_restriction_level>2 errno should contain *EINVAL*
    - On any other failure errno should contain *EINVAL*
- If there is more than one error you should return the first one by the above order.

System call number 245

```
int get_process_log(pid_t pid, int size, fai* user_mem)
```

Description

Returns in *user_mem* the last *size* records of the forbidden activities of the process with PID=*pid*.

Reading data from the log does not remove or change the log.

- The returned data should be ordered by *fai.time* (in ascending order)
- You may assume that *user_mem* has enough space to contain *size* records.
- If nothing has ever entered the log, consider it as an empty log (this is not considered as an error).
- Use the *copy_to_user* method to copy data from kernel memory to user memory.

Parameters

pid	The process for whom we would like to set the restrictions
size	The number of records we wish to read from the log
user_mem	An address in the user space into which the method should write the result

Return values

- On success: return 0
- On failure: return -1
    - If pid<0 errno should contain *ESRCH*
    - If no such process exists errno should contain *ESRCH*
    - If size> actual number of records in the log, errno should contain *EINVAL*.
    - If size<0 errno should contain *EINVAL*
    - *o* On memory copy failure errno should contain *ENOMEM*
    - *o* On any other failure errno should contain *EINVAL*
- If there is more than one error you should return the first one by the above order.

Assignment Constraints and Grading Policy

- 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.
- The assignment will be graded by auto tests. Write your own tests and be sure that your system is working according to the above specification.
- 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

Informative Notes and Technical Details

What should you do?

Use VMware, like you learned in HW0, in order to make the following changes in the Linux kernel:

1. Make necessary changes in file **kernel/fork.c, kernel/exit.c, include/linux/sched.h** and **arch/i386/kernel/entry.S**
2. Put the implementation of your logic and system calls in **kernel/hw1_syscalls.c** and **arch/i386/kernel/restrict.c** that you will have to create and add to the kernel.
3. Update the makefiles in those directories to compile your new file too. (Tip: add it to obj-y).
4. Figure out where is the central point in which every system call invocation must go through, and understand how to add your logic there.
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. Update more files if needed.
7. Recompile and run the new kernel like you did in the preliminary assignment.
8. Put the wrapper functions in **hw1_syscalls.h.** Note that **hw1_syscalls.h** is not part of the kernel, and the user should include it when using your system calls. This also means that you don't need to recompile the entire kernel when modifying the header.

9. Boot with your new Linux and try to compile and run the test program to make sure the new system calls work as expected.
10. Did it all? Good work! Submit your assignment. Submit the following: **kernel.tar.gz**, **submitters.txt** and **hw1_syscalls.h** (see below)

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 recitations.
- **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.
- Write your own tests. We will check your assignment with our test program
- Linux is case-sensitive. entry.S means entry.S, not Entry.s, Entry.S or entry.s.
- 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
- You can use existing kernel data structures without implementing them.

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 **hw1**, put them in the **hw1** 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/R7n5YjsqO8XvR8m03

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