Homework 1Wet

Due Date: 17/11/2016 12:30

Teaching assistant in charge:
- Arie Tal

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 mission in this assignment will be to add a new system calls to the interface of the kernel, and to update some existing system calls. While doing so you will gain extra knowledge in compiling the kernel.

Also 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

In this assignment you are going to add new system calls to the linux kernel that will allow inspecting and modifying process parent-child relationships.

Detailed Description

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

Note: 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.

Code Wrappers

int attach_proc(pid_t PID)

Description

Force process PID to become the youngest child process of the current process.

Prerequisites:

Current process owner must be root, or the same as that of attached process.

After successfully attaching a process, future calls from its original parent to wait() - if it was an only child, and waitpid(PID) should fail, and the new parent is responsible for waiting for the attached process.
Return values

- On success: Returns the process ID number of the former parent of process PID.
- On failure:
  - If the process PID does not exit, returns -ESRCH.
  - If process PID is the current process or any of its ancestor processes, returns -EINVAL.
  - If current process owner isn’t root, nor the same as that of attached process, return -EPERM
  - If the parent process of process PID is already waiting due to the wait() or waitpid() calls, return -EBUSY

**int get_child_processes(pid_t* result, unsigned int max_length)**

**Description**
Scans and collects the PIDs of the child processes of the current process from the youngest child to the oldest, stopping at max_length or the oldest child, whichever comes first.

*Note: In the context of this assignment, an “older” process can become a youngest child if it gets attached to a different parent.*

**Return values**

- On success, the result array that is passed will be filled with PID numbers of the child processes of the current process, starting from the youngest child. Return value should be the number of actually filled PIDs.
- On failure:
  - If the parameter result array is too small, or an invalid memory address returns -EFAULT.
    - Note: This can be easily determined by the result of copy_to_user().

**Int get_child_process_count()**

**Description**
Calculates the number of direct child processes that the current process has.

**Return values**
Always succeeds. Returns the number of child processes for the current process.
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.

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.
%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:

**Important Notes and Tips**

- Save the access log for a given process in the process descriptor struct task_struct, Its definition can be found in **sched.h**
- 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.
You should use the functions copy_from_user, copy_to_user, etc. in order to copy data from user space to kernel space and vice versa. The dry part will show you why these functions are critical in such cases. Useful link: [User space memory access from the Linux kernel](#) (skip to Kernel APIs section)

- All your changes must be made in the kernel level.
- Submit only modified files from the Linux kernel.
- You should not print the code.
- Start working on the assignment as soon as possible. The deadline is final, NO postponements will be given, and high load on the VMWare machines will not be accepted as an excuse for late submissions.
- `-ENOENT` stands for 'minus ENOENT'
- If there are more than one error when calling a syscall you should return the first one by the order in the function description.
- Don't forget to set initialize all your data structures.
- Write your own tests. We will check your assignment also 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 release all the allocated memory for the process when it ends. Look at the function do_exit
- 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.
- You may want to look on `/usr/src/linux-2.4.18-14custom/include/linux/list.h` and tutorial 3 for already implemented list.
- If you need global variables, you can put them in kernel/blocker.c (The file with your syscalls)
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. Put the implementation of the new system calls in the file kernel/pattach.c that you will have to create and add to the kernel. Update the makefile in that directory to compile your new file too. (Tip: add it to obj-y).
2. Update entry.S (add system call numbers and references in the syscall table)
3. 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.
4. Make necessary changes in file fork.c and exit.c
5. Update more files is needed.
6. Recompile and run the new kernel like you did in the preliminary assignment.
7. Put the wrappers functions in pattach.h
8. Boot with your new Linux, and try to compile and run the test program to make sure the new system calls work as expected.

Did it all? Good work, Submit your assignment.

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
```

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
```

c. A file named `pattach.h` that contains the implementation of your wrapper functions and other declarations.

**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  
|  |  +- pattach.h
```
Have a Successful Journey,

The course staff