Homework 1 Wet

Due Date: 30/4/2017 23:00

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 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

As you have seen, when a process dies it becomes a zombie. If its father invokes *wait* the zombie will be collected and its Process Control Block (PCB) will be released, But if the father process doesn't call *wait*, the zombie's PCB will stay in memory at least until the father dies. So if some process forks a lot and doesn't wait for it's zombies a kind of memory overflow will occur.
We would like to add a new feature to our linux kernel so:

1. We will able to define a zombie limit for each process, if it passes the limit it won't be able to fork till some zombies are released.
2. We would like to add an alternative way to release zombies:
   A process may request transfer some of its zombie processes.

***NOTE*: We aim to "punish" a process who is too lazy to release its zombies.**
For example: a process who had not reached its limit may fork as much as it wants - as long as it doesn't exceed its zombie's limit.
If the process exceeds its limit it won't be able to fork any more but more of its childrens may become a zombies.
Therefor the following scenario is acceptable:

1. **Set the limit to be 0**
2. **Fork 100 times**
3. **One child has become a zombie and the father doesn't wait for it.**
4. **Fork fails**
5. **All the other children became zombies**
6. **The limit was set to 0 but the zombie's number is 100**

Detailed Description

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

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. ***In order to make the ease the task at hand, the new feature should work only if the limit was set.***
   For example: The Init process should not be affected by your changes (because you can't set a limit for it)

Code Wrappers

int set_max_zombies(int max_z, pid_t pid)

Description

Sets a limit of *max_z* on the allowed number of zombies of the process with pid=*pid*.
After setting a limit, if the process exceeded it and tries to fork, the fork call should fail (return *-ENOMEM*).

Return values

- On success: Return 0.
- On failure:
  - If *max_z* < 0, return -1 and *errno* should contains *EINVAL*.
  - If pid < 0 return -1 and errno should contains *ESRCH*

int get_max_zombies()

Description

Get the zombies limit of the calling process.

Return values

- On success: return the zombies limit
- On failure: if the limit is undefined return -1 and *errno* should contain *EINVAL*.

int get_zombies_count(pid_t pid)

Description

Return the current number of zombie processes with pid=*pid* (The zombies that are child-processes of the process with the corresponding pid)

Return value

- On success: return the required number, if no limit was defined return 0
- On failure:
  - If pid < 0 return -1 and errno should contains *ESRCH*

pid_t get_zombie_pid(int n)

Description

Return the pid of the nth (starting with 0) zombie process among the children of the calling process.

The children pids should return in the following order:
The first process to become a zombie is the 1st, the next is the 2nd and so on.
For example: assume *p* forked *p1* with *pid1*, *p2* with *pid2* , and *p2* became zombie before *p1*.
then:
calling get_zombie_pid(0) will return *pid2*
and calling get_zombie_pid(1) will return *pid1*

Return value

- On success: return the required pid.
- On failure:
  - If *n* is bigger than or equal to the zombies number of the calling process: return -1 and *errno* should contain *ESRCH*.
  - If no limit was defined treat it as if the zombies number is 0. (return -1 and errno should contain *EINVAL*)

int give_up_zombie(int n, pid_t adopter_pid)

Description

Ask from the process with pid = *adopter_pid* to adopt the nth first zombies of the calling process according to the order mentioned above.
The adopter state should stay correct.
Meaning:

1. If a process adopted zombies, calling to *get_zombies_count* with its pid should return the new number of zombies.
2. The adopted zombies should stay in their original order, and should be placed right after the adopter's zombies.

Note: if the adopter's limit was undefined it can't adopt any zombie.

Return value

- On success: return 0.
- On failure: return -1
    - If *n* is bigger than the zombies number of the calling process *errno* should contain *EINVAL*.
    - If *n* + adopter's *zombies_num* > *adopter_limit* then *errno* should contain *EINVAL*.
    - If the adopter's limit is undefined return *EINVAL.*
    - If pid < 0 errno should contains *ESRCH.*

Clarifications

- Each process starts with no limit on its zombies.
  for example: if we set a limit of 10 zombies to p1, and p1 forked p2, so p2 should have no limit on its zombies.
- *wait* and *give_up_zombies* should not contradict each other.
  for example: assuming we have 100 zombies, and by using wait we released 50, then *give_up_zombie* can give at most 50 zombies. (*the order should stay the same but without the collected zombies*)
- if no limit was defined, you don't need to save any data (zombie's order, zombie's number etc.)
- If a process already has zombies when calling *set_max_zombies,* you can ignore them, ***Meaning***: you should count and order only zombies which were created after the limit was set.
- Assume that the limit will be set only once for each process.

- Calling give_up_zombies with the pid of the calling process should work as described in the function. (Meaning the process takes the N first zombies and moves them to the end of its own zombie list)
- If the process has zombies while it dies, its zombies should move to Init (as usual).

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.

```
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:

```
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:

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
http://www.ibm.com/developerworks/linux/library/l-ia/index.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. Put the implementation of the new system calls in the file **kernel/syscalls_zombies.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 if needed.
6. Recompile and run the new kernel like you did in the preliminary assignment.
7. Put the wrappers functions in **syscalls_zombies.h,** note that **syscalls_zombies.h** is not part of the kernel, the user should include it when using your system calls.
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.
9. Submit **kernel.tar.gz**, **submitters.txt** and **syscalls_zombies.h** (see below)

Did it all? Good work, Submit your assignment.

Important Notes and Tips

- First, try to understand exactly what your goal is.
- Think which data structures will serve you in the easiest and simplest way, (*Hint: you might have seen them in the tutorials).*
- You are supposed to handle zombies, notice to what should happen when a zombie is released, do you need to update its parent's data?
- 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.
- The linux developers wrote comments in the code, read them, they might help you to 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.
- All your changes must be made in the kernel level.
- Submit **syscalls_zombies.h** and 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 a 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 an already implemented list.
- If you need global variables, you can put them in kernel/syscall_zombies.c (The file with your syscalls)

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
```

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
syscalls_zombies.h
```

The zip should look as follows:

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

Have a Successful Journey,
The course staff