Assignment 2: Syscalls[1]

Notes

- This assignment is to be done **individually or in pairs**. Do not share your code or solution, do not copy code found online, ask all questions on Piazza discussion forum (see webpage).
  - You may use code provide by the instructor, or in the guides/videos provided by the instructor (such as instructor's YouTube videos).
  - You may follow any guides you like online as long as it is providing you information on how to solve the problem, as opposed to a solution to the assignment.
  - If receiving help from someone, they must be helping you debug **your** code, not sharing their code or writing it for you.
  - We will be carefully analyzing the code submitted to look for signs of plagiarism so please don't do it! If you are unsure about what is allowed please talk to an instructor.
- You may not resubmit code you have previously submitted for any course or offering.
- Instructors and TAs may conduct followup interviews with students to discuss their implementation and design in order to verify the originality of the assignment.

1. Preparation

In order to complete this assignment, it is vital that you have carefully completed and understood the content in the following guides which are posted on the course website:

- **Custom Kernel Guide**: how to download, build, and run a custom Linux kernel.
- **Guide to Linux Syscalls**: how to create and test a simple Linux system call (syscall).

In this assignment you'll be coding in both user space and kernel space. Since it takes a couple of minutes to recompile and re-run a new kernel, you should code carefully!

2. Array Statistics Syscall

First, you'll add a new system call that computes some basic statistics on an array of data. In practice it makes little sense to have this as a syscall; however, it allows us to become familiar with accessing memory between user and kernel space before accessing other kernel data structures.

2.1 Syscall Requirements

- Create a new syscall named `array_stats`:
  - Make its syscall number 549 (in `syscall_64.tbl`):

```
549   common      array_stats       sys_array_stats
```

  - The syscall will compute some simple statistics about an array of data. It will write these statistics to a struct. Your syscall will need to be passed:
    - a pointer to the struct for the syscall to write the statistics to
    - a pointer to the array of data for the syscall to analyze

1 Based on an assignment by Mohamed Hafeeda.

- the size of the array of data (i.e., how much data is there)

- In your .c file which implements the syscall, you'll need:

```
struct array_stats_s {
        long min;
        long max;
        long sum;
};

SYSCALL_DEFINE3(
        array_stats,                    /* syscall name */
        struct array_stats_s*, stats,   /* where to write stats */
        long*, data,                    /* data to process */
        long, size)                     /* # values in data */
{
        // your code here…
}
```

  - `array_stats`: The first argument to the macro defines the name of the syscall function. The macro will make this into the function `sys_array_stats(…)`.
  - `stats`: A pointer to one `array_stats_s` structure which was allocated by the user-space application. Structure will be written to by the syscall to store the minimum, maximum, and sum of all values in the array pointed to by `data`.
  - `data`: An array of `long` values passed in by the user-space application.
  - `size`: The number of elements in `data`. Must be > 0.

- The `array_stats` syscall must:
  - Set the three fields of the `stats` structure based on the `data` array. The values in `data` are signed (positive or negative). Nothing special need be done if the sum of all the data will overflow/underflow a `long`.
  - Return 0 when successful.
  - Returns `-EINVAL` if `size` <= 0 (in which case, nothing is written to `stats`).
  - Returns `-EFAULT` if unable to access `stats` or `data` (such as invalid pointers) (in which case, nothing is written to `stats`).
  - You must *not* allocate or use a large amount of kernel memory to copy the entire input `data` array into.

The Guide to Linux Syscalls (on course website) has some hints on how to write your own syscall.

2.2 Test Application Requirements

You must create a user-space test application to test your syscall.

- Code your application in a folder outside of the kernel's build folders. For example, put it where ever you normally put your code for assignments.
- Your application should execute a few calls to your syscall to prove it works. Try some of the edge cases.

- Check with different size arrays, and different values.
- Check with invalid pointers and array sizes to ensure your syscall handles bad values correctly.

- We will test your code with our extensive testing program.

3. Process Ancestor Syscall

In this section you'll implement a syscall which returns information about the current process, plus its ancestors (its parent process, it's grandparent process, and so on).

3.1 Syscall Requirements

- Create a new syscall named `process_ancestors`:
  - Make its syscall number 550 (in `syscall_64.tbl`):

```
550    common      process_ancestors sys_process_ancestors
```

  - The syscall will look at the current process, and each of its ancestors, all the way back to the `init` process (usually PID 1). For each process it will record some information about the process. This infromation will be written into an array of structs (one per process). Our syscall will need:
    - a pointer to an *array of structs* for recording the process information, allocated by the user-space application.
    - the number of structs in the array
    - a pointer to a `long`, which the syscall will write to, to record how many processes the syscall analyzed (i.e., how many elements in the array of stucts did the syscall fill).
  - In your syscall's .c file you'll need:

```
#define ANCESTOR_NAME_LEN 16
struct process_info {
      long pid;                     /* Process ID */
      char name[ANCESTOR_NAME_LEN]; /* Program name of process */
      long state;                   /* Current process state */
      long uid;                     /* User ID of process owner */
      long nvcsw;                   /* # voluntary context switches */
      long nivcsw;                  /* # involuntary context switches */
      long num_children;            /* # children process has */
      long num_siblings;            /* # sibling process has */
};

SYSCALL_DEFINE3(
      process_ancestors,                /* syscall name for macro */
      struct process_info*, info_array, /* array of process info strct */
      long, size,                       /* size of the array */
      long*, num_filled)                /* # elements written to array */
{
      // your code here…
}
```

- `process_ancestors`: the first argument to the macro defines the name of the syscall function. The macro will make this into the function `sys_process_ancestors(…)`.
- `info_array`: array of `process_info` structs, alloctaed by the user-space application, into which the syscall will write information about each process.
- `size`: the size of `info_array` (number of `process_info` structs), passed in by the user-space application. This is the maximum number of structs that the syscall will write into the array (starting with the current process as the first entry and working up from there). The `size` may be larger or smaller than the actual number of ancestors of the current process: larger means some entries are left unused; smaller means information about some processes will not be written into the array.
- `num_filled`: a pointer to a `long`, which will be written to by the syscall to record how many structs in the `info_array` array were filled in by the syscall. `num_filled` will be at most equal to `size`; but could be less than `size` when there are fewer ancestors processes than the size of the passed in array.

- The `process_ancestors` syscall must:
  - Starting at the current process, fill the elements in `info_array[]` with the correct values.
    - Ordering: the current process's information goes into `info_array[0]`; the parent of the current process into `info_array[1]`; grandparent into `info_array[2]`; and so on.
    - If there are extra structs in `info_array[]`, they are left unmodified.
  - Return 0 when successful.
  - Return `-EINVAL` if the syscall's `size` argument is <= 0.
  - Return `-EFAULT` if there are any problems access `info_array` or `num_filled` (such as bad pointers, ...)
  - You must *not* allocate or use a large amount of kernel memory to copy/store large arrays into. However, you may allocated on the stack, inside the syscall, up to one `process_info` struct if needed.

3.2 Test Application Requirements

You must create a user-space test application to test your syscall.

- Place the test apps for both of your assignment's system calls in one folder, built with the same makefile.
- You must do at least some testing to show the syscall works correctly, and that it generates correct error values in at least a few of the failure conditions (suc has bad pointers, ...).
  - Hint: Use asserts or the CHECK() macro in your test code.
- We will run your test apps to ensure they work.
  We will test your syscall with our extensive testing program.

3.3 Hints

- You will make extensive use of the kernel's `task_struct` structures: each process has a `task_struct`. You can access the `task_struct` for the current process using the macro

`current`.

- For example, the PID for the currently running process on this CPU can be found with:
  `long pid = current->pid;`

- Basic algorithm sketch:
  1) Fill in fields for `info_array[0]` for the current process.
  2) Move to the parent of this process (`current->parent`), and copy its info into `info_array[1]`.
  3) Repeat until the parent of the process you are working on is itself
  (i.e: `cur_task->parent == cur_task`).

  - The first task spawned by Linux (often called `init`) is its own parent, so hence its `parent` pointer points to itself. This process usually has PID 1 and is the idle task (named `swapper` or `init`).
  - I recommend that you:
    - first write test values (like 42) into the `process_info` struct to show you can correctly write to it in the syscall, and read from it in your user-space test application.
    - next get the info on the current process and print it to the screen (`printk`) to ensure you have the correct values.
    - then write the process's values into the `process_info` struct and handling ancestors.

- Hints on filling the fields of `process_info`:
  - Quite a few of the values can be pulled directly out of the `task_struct` structure. Look for fields with a matching name.
    - `task_struct` is defined in `include/linux/sched.h` (in your kernel folder). To include this in your syscall implementation use: `#include <linux/sched.h>`
    - Here is a good online site to navigate the kernel source;
      browse into `include` → `linux` → `sched.h`; search for `tast_struct`
  - The name of the program for a process is stored in the `task_struct.comm` field.
  - The user ID for the owner of a process can be found inside the process's credentials (`cred` field). Inside `cred`, you want to look at the `uid` field.
  - For counting the number of children and siblings, you'll want to start with the following linked list nodes: `task_struct.children`, and `task_struct.sibling`.
    - These are nodes in circular linked lists. Linux uses the struct `list_head` for a node because in a circular linked list, each node can be thought of as the head of the list.
    - You can follow the `next` field of a node in the list (a `list_head`) to get the node (`list_head`) of the next element in the list.
    - It is a circular linked list, so you'll have to determine how to count the number of elements (think of how you know when to stop following next pointers).
    - Note that Linux has some clever (complicated) ways of taking a node in the list (which just has a `next` and `prev` field pointing to other `list_head` structures) and accessing the full structure that contains the node. For example, given a `list_head struct` that is in a

`task_struct`, the kernel includes macros to give you the full `task_struct`. However, you have (mercifully) been spared having to do this. If you are interested, for fun try printing out the PID of each of the sibling processes.

- Safe memory access is critical. Apply all the suggestions from the syscall guide for safe memory access.
- You can use the `ps` command in your QEMU virtual machine to display information on running processes and verify the syscall output. Run '`man ps`' to see how to select the information it displays.

4. Deliverables

In CourSys, create a group for your submission. Even if you worked on your own, you need to be in a group (of 1 in this case) in order to submit. Submit the following to CourSys:

1. `syscalls.tar.gz`: An archive file of your kernel code folder.
   Required files:
   - `makefile` (executed by kernel's build system to build your syscalls' `.c` files)
   - array stat syscall's `.c`
   - process ancestor syscall's `.c`
   - OK to include `cs300_test.c`
   - OK to include any extra `.h` files (if any)

   To create this archive, from inside your `linux-5.7/cs300/` folder, execute:
   ```
   $ tar -czvf syscalls.tar.gz *
   ```

2. `user-tests.tar.gz`: An archive file of your user mode folder containing your test code for the syscalls. Must include a `makefile` with targets "`all`" (to build) and "`transfer`" (to copy the statically linked executable to QEMU via SCP on port 10022).
   May include the provided `array_statistics` test code.

   From inside your user-space folder, execute:
   ```
   $ tar -czvf user-tests.tar.gz *
   ```

3. Screenshot of QEMU running your custom kernel.
   - Launch your custom kernel in QEMU which you customized from the guide to include your SFU user ID.
   - Compile and transfer in the provided `cs300_test` test.
   - Run the comamnd:
     ```
     # uname -a
     ```
   - Run the command
     ```
     # ./cs300_test
     ```
   - Take a screenshot of the QEMU terminal showing at least the output of these two commands (OK if there are extra lines on the screen, and OK if your syscall's printk() messages appear too.

Please remember that all submissions will automatically be compared for unexplainable similarities.

Marking will be done on a 64-bit system. If you did your development on a 32-bit system, your code must still compile and run on a 64 bit system. In this case, you may find the following user-level code useful to define the syscall numbers based on the system's 32bit vs 64bit status (the constant _LP64):

```
// Test application to call the CS300_TEST syscall
// Syscall is expected to take one argument and
// return that value + 1.

#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

// Syscall numbers
#if _LP64 == 1
       // 64 bit
       #define SYSCALL_CS300_TEST       548
       #define SYSCALL_ARRAY_STATS      549
       #define SYSCALL_PROCESS_ANCESTOR 550
#else
       // 32 bit
       #define SYSCALL_CS300_TEST       439
       #define SYSCALL_ARRAY_STATS      440
       #define SYSCALL_PROCESS_ANCESTOR 441
#endif

int main(int argc, char *argv[])
{
       printf("\nDiving to kernel level\n\n");
       int result = syscall(SYSCALL_CS300_TEST , 12345);
       printf("\nRising to user level w/ result = %d\n\n",
              result);
       return 0;
}
```