Project 4: System Calls[1]

1. Preparation

To complete this assignment, it is of paramount importance 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 Sys-Calls**: how to create and test a simple Linux system call (sys-call).

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!

You can work in pairs. You can also work alone, but you get **no** extra credits for this.

2. Array Statistics Sys-Call

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 sys-call; however, it allows us to become familiar with accessing memory between user and kernel space before accessing other kernel data structures.

Requirements

In the kernel's `cs300/` folder, create a file named `array_stats.h` with the contents:

```
// Define the array_stats struct for the array_stats sys-call.
#ifndef _ARRAY_STATS_H_
#define _ARRAY_STATS_H_

struct array_stats{
        long min;
        long max;
        long sum;
};

#endif
```

Create a new sys-call named `array_stats` (function `sys_array_stats()`):

- Implement it in your kernel's `cs300/` folder in a file name `array_stats.c`. Use `#include "array_stats.h"`
- Make it sys-call number 341 (in `syscall_64.tbl`).
- The sys-call's prototype is:

```
asmlinkage long sys_array_stats(
                struct array_stats *stats,
                long data[],
                long size);
```

[1] Created/updated by Brian Fraser and Mohamed Hefeeda, updated by Julian Rrushi.

- `stats`: A pointer to one `array_stats` structure allocated by the user-space application. Structure will be written to by the sys-call to store the minimum, maximum, and sum of all values in the array pointed to by `data`.
- `data`: An array of `long int` values passed in by the user-space application.
- `size`: The number of elements in `data`. Must be > 0.

The `array_stats` sys-call 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 overflows/underflows a `long`.
- Return 0 when successful.
- Returns `-EINVAL` if `size` <= 0.
- Returns `-EFAULT` if any problems access `stats` or `data`.
- You must *not* allocate or use a large amount of kernel memory to copy the entire input `data` array into.

Hints

- Use `printk()` calls in the kernel to print out debug information. You *may* leave some of these `printk()` messages in your solution as these messages are not technically displayed by the user-space application. The messages you leave in should be helpful such as showing parameters values or errors it caught; they should not be of the sort "running line 17", or "past loop 1". It is usually useful to `printk()` the parameters you are given, and `printk()` any error conditions you handle.
- Correct memory access is the hardest part of writing this sys-call.
  - The kernel cannot trust anything it is given by a user-level application. Each pointer, for example, could be: a) perfectly fine, b) null, c) outside of the user program's memory space, or d) pointing to too small an area of memory. Since the kernel can read/write to any part of memory, it would be disastrous for the kernel to trust a user-level pointer.
  - Each read you do using a pointer passed in as input (a user-level pointer) should be done via the `copy_from_user()` macro. This macro safely copies data from a user-level pointer to a kernel-level variable: if there's a problem reading or writing the memory, it stops and returns non-zero without crashing the kernel.
    - First use this macro to copy values into a local variables (which are in the kernel's memory space). Then, use these local variables in your program. See the Linux Kernel Development (ch5, p75) for more on the macro.
    - Hint: Create a local variable of type `long` inside your sys-call function. Use `copy_from_user()` to copy one value at a time from the user's `data` array into this variable. If a copy fails (`copy_from_user()` returns non-zero) then have your sys-call end immediately and return `-EFAULT`.
    - Hint: Double check that you only ever access the `data` array using `copy_from_user()`!
    - You *can* directly access `size` because it is passed by value so there is no possible problem access memory.

- Likewise, when writing to a user-level pointer, use `copy_to_user()` which checks the pointer is valid (non-null), inside the user-program's memory space, and is writable (vs read-only).
  - Hint: Create a local variable of type `struct array_stats` inside your sys-call function. Compute the correct values in this `struct` first, then at the very end use `copy_to_use()` to copy the contents to user's pointer.
- The kernel is compiled with C90; you must declare your variables at the start of a block (such as your function) vs in the middle of your function.
- A user-space test program is provided to extensively test your system call. Your sys-call implementation should pass all of these tests. It is likely that your code will be marked based on passing these (and perhaps other) tests. You may want to run the tests one at a time by commenting out calls in `main()`.

3. Process Ancestor Sys-Call

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

Requirements

- In the kernel's `oscourse/` folder, create a file named `process_ancestors.h` with the contents:

```
// Structure to hold values returned by process_ancestors sys-call
#ifndef _PROCESS_ANCESTORS_H
#define _PROCESS_ANCESTORS_H

#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 */
};

#endif
```

- Create new sys-call named `process_ancestors` (function `sys_process_ancestors()`):
  - Implement it in your kernel's `oscourse/` folder in a file name `process_ancestors.c`. Use `#include "process_ancestors.h"`
  - Make its sys-call number 342 (in `syscall_64.tbl`).
  - The sys-call's prototype is:

```
asmlinkage long sys_process_ancestors(
                        struct process_info info_array[],
                        long size,
                        long *num_filled)
```

  - `info_array[]`: An array of `process_info` structs that will be written to by the kernel as it fills in information from the current process on up through its ancestors.
  - `size`: The number of structs in `info_array[]`. This is the maximum number of structs that the kernel 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 (see `num_filled`); smaller means information about some processes not written into the array.
  - `num_filled`: A pointer to a `long`. To this location the kernel will store the number of structs (in `info_array[]`) which are written by the kernel. May be less than size if the number of ancestors for the current process is less than `size`.

- The `process_ancestors` sys-call 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.
    - Extra structs in `info_array[]` are left unmodified.
  - Return 0 when successful.
  - Returns `-EINVAL` if `size` <= 0.
  - Returns `-EFAULT` if any problems access `info_array` or `num_filled`.
  - You must *not* allocate or use a large amount of kernel memory to copy/store large arrays into.
- You must also create a user-space test program which calls your sys-call and exercises its functionality. You must do at least some testing on to show it works correctly, and that it generates correct error values in at least a few of the failure conditions (bad pointers, ...).
  - Hint: Use asserts in your test code.
  - We will have an extensive test suit to exercise your solution!

Hints

- You will make extensive use of the kernel's `task_struct` structures: each process has a `task_struct`. You can find 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) Start from current process and fill in fields for `info_array[0]`.
  2) Move to 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 (`cur_task->parent == cur_task`).

- The first task spawned by Linux is its own parent, so hence its `parent` pointer points to itself. This process has PID 0 and is the idle task (named swapper).
- I recommend that you first get the info on the current process and print it to the screen (`printk`) to ensure you have the correct values. Then work on getting the data into the `process_info` structs and handling ancestors.

- Hints on 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 sys-call implementation use: `#include <linux/sched.h>`
    - Here is a good online site to navigate the kernel source.
  - 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). Hint: Think of addresses.
    - 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 previous sys-call for safe memory access.
- You can use the `ps` command in your QEMU virtual machine to display information on running processes and verify the sys-call output. See ps's man page for how to select the information it displays.

4. Deliverables

Submit the following material on Canvas:

1. A patch of your Linux kernel folder for the files you changed.
   Patch must be done in accordance with the Kernel Patch guide posted on the course website.
   Must patch cleanly (either using **`Git`** or **`patch`**).
2. An archive file (zip or tar.gz) of your kernel code:

- oscourse/ folder
- /makefile, syscall_64.tbl

3. An archive file (zip or tar.gz) of your user mode folder containing your test code for the sys-calls. Must include a makefile with a target "transfer" to copy the statically linked executables to QEMU.
   May include the provided `array_statistics` test code.

- 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 perfectly on a 64 bit system. Specifically, double check that you have:

1. Created the necessary rows in `arch/x86/syscall/syscall_64.tbl`
   These will allow us to call your functions, but the numbers must match exactly those numbers listed in this assignment. (In 32-bit the numbers differ from 64bit).
2. Your user-level code uses the correct 64-bit sys-call numbers.
   Hint: Make your user-level code check the GCC defined constant `_LP64` and automatically pick the correct sys-call number:

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

#if _LP64 == 1
#define _CS300_TEST_ 340
#else
#define _CS300_TEST_ 390
#endif

int main(int argc, char *argv[])
{
        printf("\nDiving to kernel level\n\n");
        int result = syscall(_CS300_TEST_ , 12345);
        printf("\nRising to user level w/ result = %d\n\n", result);

        return 0;
}
```