Project 3: Linux System Calls

CS 311

For this project, you'll implement three Linux kernel syscalls. You'll also implement a small wrapper shared library to present a cleaner API to users of your syscalls. For each of the syscalls, you'll need to add a unique syscall number to `unistd_32.h` and a syscall table entry to `syscall_table_32.S`, just as you did in Project 1. The three syscalls themselves can be added to your `lab_syscalls.c` file from Project 1.

Once again, you'll be working from your Linux virtual machines, using your GitHub repository from the syscall exercise of Project 1.

References

The following are all available on, or linked to from, the course web site:

1. Program Library HOWTO, Chapter 3 Shared Libraries. Recommended reading for those curious about the background and details of shared libraries in Linux.
2. `libmysyscall.h` — Syscall API declaration file. Needed by both your library source code and any user program using your library.
3. `libmysyscall.c` — Syscall API definition file. This will be compiled into the shared library.
4. `semaphore.h` and `semaphore.c` — The kernel source files for kernel semaphores. See below for usage examples.

`libmysyscall` Shared Library

You can implement this as soon as you've added the three unique syscall numbers to `unistd_32.h` and added the three corresponding syscall entries to `syscall_table_32.S`. See `libmysyscall.c` for the numbers to use.

This library will allow users of your syscalls to invoke them like this:

```
myprintint(val);
```

rather than this:

```
syscall(__NR_tpk_sysc1, val);
```

Download `libmysyscall.h` and `libmysyscall.c` into a directory in your kdev account (somewhere under your kernel source directory would be a good idea, so that you can put the files under revision control).

You'll need to edit one line in `libmysyscall.c`.

Follow the recipe below for installing `libmysyscall.h`. Then, build the shared library and install it.

Here's the recipe for building and installing the header file and shared library, along with instructions for using the shared library in a user program:

```
# cd to whatever directory contains libmysyscall.c and libmysyscall.h

cd ~/<directory containing library files>


# Install libmysyscall.h.  Repeat any time changes are made to
# libmysyscall.h.

sudo cp libmysyscall.h /usr/include


# _All_ of the following steps have to be repeated any time changes are
# made to either libmysyscall.c or libmysyscall.h.


# Create libmysyscall.o

gcc -fPIC -g -c -Wall libmysyscall.c


# Creates libmysyscall.so.  Note _carefully_ the arguments to the "upper W
lower el" switch and that there are _no_ whitespace characters in this
argument list.  And, that's a zero in the middle of the line, not a
capital Oh.

gcc -shared -Wl,-soname,libmysyscall.s0 -o libmysyscall.so libmysyscall.o


# Move libmysyscall.so to /usr/lib

sudo cp libmysyscall.so /usr/lib


# Make the shared library file known to the linker

sudo /sbin/ldconfig


# Check that all is fine

/sbin/ldconfig -p | grep libmysyscall

# (The linker's cache entry to your shared library file should be printed.)
# Any source program using this library needs to contain the line:

   #include <libmysyscall.h>

# and be explicitly linked with the library:

gcc -o timeTest timeTest.c -lmysyscall
```

You won't be able to successfully run any program using your syscalls until you've built and booted a kernel containing the syscalls.

Current Time Syscall

Implement a syscall which returns the value of the kernel's `xtime` variable to the user process. `xtime` is of type `struct timespec`, defined as:

```
struct timespec {
   time_t   tv_sec;   /* seconds */
   long  tv_nsec;     /* nanoseconds */
};
```

The signature of your syscall should be

```
asmlinkage int sys_lab_sysc2(struct timespec *ct)
```

If the user process passes in a memory address to which it does not have write permission or the syscall can't copy the value of `xtime` to the user process, the syscall should return -1. Otherwise, it should return 0. The syscall should log each of its invocations to the system log.

`xtime` access requires multiple memory accesses. Because of this, your syscall could be interrupted at the boundary of any of these accesses when it is reading `xtime` from the kernel to its own storage. During this time, the value of `xtime` could be changed by the kernel, thus invalidating the value you have already partially read. Therefore, synchronization primitives need to be used when reading `xtime`. See below.

Here's pseudo-code for your syscall:

```
asmlinkage int sys_lab_sysc2(struct timespec *ct)
{
   confirm write access to the struct timespec pointed to by ct,
      or return -1;

   perform a synchronized copy of xtime into a local struct timespec
      variable;

   copy the local struct timespec to user space, returning -1
      if unsuccessful;

   return 0;
}
```

You can use the userland program `timeTest.c` to test your syscall.

Producer/Consumer Problem Syscalls

Implement syscalls for producing/consuming a single 1024 `char` array into a buffer of four such arrays. The signatures for your syscalls should be:

```
asmlinkage int sys_lab_sysc3(char *str)   /* producer */
asmlinkage int sys_lab_sysc4(char *str)   /* consumer */
```

These syscalls should return 0 on success, otherwise -1. These syscalls should also log each of their invocations to the system log. I would suggest making use of the following global variables:

```
#include <linux/semaphore.h>

#define MAX_LENGTH 1024
#define MAX_COUNT 4

static struct semaphore lock = __SEMAPHORE_INITIALIZER(lock, 1);
static struct semaphore full = __SEMAPHORE_INITIALIZER(full, 0);
static struct semaphore empty = __SEMAPHORE_INITIALIZER(empty, MAX_COUNT);
static char mesg[MAX_COUNT][MAX_LENGTH];
static int in = 0;
static int out = 0;
```

Assume that there will be multiple producers and multiple consumers. You can find pseudo-code for this problem in the textbook in the *Classic Problems of Synchronization* Section. Do **not** implement the `do/while` loops shown in the textbook — looping will be handled within the userland process.

You'll need to handle copying data between user space and kernel space, as well as ensuring that you have read or write access, as appropriate. You'll also need to use kernel level semaphores. See below for documentation and examples.

You can use the userland programs `produce.c`, `consume.c` and `multiple.c` to test your syscalls.

Userland Memory Access Functions

The following are only necessary for copying memory data between user space and kernel space; i.e., when a pointer is passed to a syscall. These aren't needed for pass-by-value parameters.

1. Validating read or write access to user memory block:

   ```
   access_ok(type, addr, size)
   ```

   Returns true (non-zero) is access is allowed, otherwise returns false (0). `type` should either be `VERIFY_READ` or `VERIFY_WRITE`. `addr` is a pointer to the first byte of the memory block to be tested. `size` is the size, in bytes, of the memory block to be tested. Example:

   ```
   if (!access_ok(VERIFY_READ, ptr, sizeof(struct timespec)))
      return -1;
   ```

2. Copying a memory block from user space to kernel space:

   ```
   copy_from_user(to_ptr, from_ptr, size)
   ```

Copies `size` bytes from user space memory block beginning at `from_ptr` to kernel space memory block beginning at `to_ptr`. Returns number of bytes that could not be copied. On success, this will be 0.

3. Copying a memory block from kernel space to user space:

   ```
   copy_to_user(to_ptr, from_ptr, size)
   ```

   Copies `size` bytes from kernel space memory block beginning at `from_ptr` to user space memory block beginning at `to_ptr`. Returns number of bytes that could not be copied. On success, this will be 0.

Synchronization Code for Reading `xtime`

A sequence lock is used to synchronize readers and writers of various kernel variables, including `xtime`. `xtime_lock` is used to guard `xtime`, which can't be read in a single memory access. The following code should be used when reading `xtime`. Note that `linux/seqlock.h` must be included in your kernel program.

```
unsigned seq;
struct timespec curTime;

do
  {
    seq = read_seqbegin(&xtime_lock);
    curTime = xtime;
  } while (read_seqretry(&xtime_lock, seq));
```

Upon exit from the do/while, `curTime` will contain a coherent value of `xtime`.

Kernel Semaphore Primitives

To use these, you must add

```
#include <linux/semaphore.h>
```

to your source file.

Creating and initializing semaphores:

```
static struct semaphore sem0 = __SEMAPHORE_INITIALIZER(sem0, 1);
static struct semaphore sem1 = __SEMAPHORE_INITIALIZER(sem1, 42);
```

Your semaphores will have to be global variables. It's best to declare them `static` to limit their scope to your source file. In this example, `sem0` is created and initialized to the value 1 and `sem1` is created and initialized to the value 42.

Decrementing a semaphore:

```
if (down_interruptible(&sem0))
{
   // The semaphore wasn't acquired.  Clean up and get out.
}
```

`down_interruptible()` puts the process to sleep until it acquires the semaphore. However, if the process is interrupted while asleep, then `down_interruptible()` will returns a non-zero value, indicating that it wasn't able to acquire the semaphore. This should be treated as an error condition — undo anything that was done to this point and return a "fail" value to the user program.

Incrementing a semaphore:

```
up(&sem0);
```

What could be simpler?

Deliverables

By the project deadline, email to me the GitHub https URL of your kernel repository. The names of your team members should be listed in the README.md file in your repository. Make sure that the work that you want me to assess is in the (default) master branch of your repositories. This project will be assessed as follows:

- Current time syscall: 15%
- Producer/Consumer syscalls: 60%
- Documentation and design: 25%

As part of your documentation, your README.md file should indicate where I can find your shared library source files and your userland test programs. The design component of your assessment will address your solution's completeness in detecting and recovering from error conditions.

Remember to document assistance you received from others. This can be done in the email you send me your repo URLs, in your README.md files, or in comments at the very top of your source code files (see fixme.c for an example).