Term Project Phase I:
System Calls

Goal: You will learn how to make a system call and about reading / writing user space from / to the kernel by adding a new function to the kernel. The function itself is trivial - it simply gives information about current process.

Introduction

A system call is the name of a kernel function that is exported for use by userspace programs. Kernel functions that appear on the system call interface cannot be called directly like an ordinary function. Instead, they must be called indirectly via the trap table. Thus if you write a new kernel function, then you need to create a new entry in the kernel trap table to reference your new function. If user-space programs are to call the program like they do any other system call, then you also need to provide a system call stub function (that contains a trap instruction). Strictly speaking, you can avoid creating the system call stub by using the system_call() routine, as explained in the following section that follows.

In the remainder of these introductory remarks, you can read the details of how system calls are set up in the kernel, how a kernel function generally is organized, and how your new kernel function can read and write user-space variables. However, to do the exercise, you will probably need to explore various parts of the kernel source code (use Linux text searching tools to do this, such as find and grep).

The System Call Linkage

A user-space program calls a system call stub, which contains a trap instruction. As a result, the CPU switches to supervisor mode and begins to execute at a specified location in kernel space. In the i386 hardware, the trap instruction actually causes interrupt 0x80 to occur, with the ISR address pointing at the entry point of the system_call() assembler routine (see arch/i386/kernel/entry.S). This code uses an argument as the offset into the sys_call_table (also defined in arch/i386/kernel/entry.S). In the Version 2.2.12 source code, the table is defined as follows.
Entry 1 contains the address of the exit() system call (the kernel function named sys_exit), 2 is for fork(), and so on.

Under usual processing, the system_call() function saves the context of the calling process, checks to be sure that the function number is in range, and then calls the kernel function. The flow of control differs if kernel tracing is enabled by using the syscall_trace() function. In this case, system_call() invokes syscall_trace() before and after the function call.

The handout given (about Linux Overview) describes ret_from_sys_call processing, used to process the bottom halves of ISRs and to call the scheduler. This block of code actually appears in the arch/i386/kernel/entry.S file.

In ANSI C programs, the compiler uses function prototypes to check that a function call agrees with the target function header. The function call is compiled only if it provides the correct number and types of arguments according to the prototype definition. The system call linkage is dynamic, meaning that it does not use the compiler's type checking mechanism. Instead, when the kernel function begins to execute, it presumes that the correct number and type of arguments have been placed on the stack when the function is called. In several cases, the kernel function checks for obvious error values (such as, an attempt is made to de-reference a null pointer), but you have no assurance that bad parameter values will be caught by the kernel function.

**Defining the System Call Number**

System calls are defined in sys_call_table. Thus when you add a new system call, you need to add an entry to the table. You do this by editing the table in the arch/i386/kernel/entry.S file, as follows.

```
.data
ENTRY(sys_call_table)
   .long SYMBOL_NAME(sys_ni_call) /* 0 */
   .long SYMBOL_NAME(sys_exit)
   .long SYMBOL_NAME(sys_fork)
   .long SYMBOL_NAME(sys_read)
   .long SYMBOL_NAME(sys_write)
   .long SYMBOL_NAME(sys_open) /* 5 */
...
   .long SYMBOL_NAME(sys_signalstack)
   .long SYMBOL_NAME(sys_sendfile)
   .long SYMBOL_NAME(sys_ni_call)
   .long SYMBOL_NAME(sys_ni_call)
   .long SYMBOL_NAME(sys_fork)
   /*
   * NOTE!! This doesn't have to be exact—we just have to make sure we
   * have enough of the "sys_ni_call" entries. Don't panic if you notice
   * that this hasn't been shrunk every time we add a new system call.
   */
   .rept NR.syscall_igo
   .long SYMBOL_NAME(sys_ni_call)
   .endr
```
This editing allows a trap (interrupt 0x80) with an argument of 191 to invoke a new kernel function, **sys_my_new_call()**. Notice that by editing this file, you change your original copy of the kernel source code. Therefore you should make two copies of the original entry.**S in a user-space directory in which you are developing your solution.** Retain one copy to ensure that you have a copy of the original, and use the other as your experimental version. Edit the experimental version, and copy it into **arch/i386/kernel**, to replace the kernel version. Note, you need superuser (**su**) permission to complete this copy operation because you are placing a new version of the file in the directory that contains the kernel source code.

This new system call can be invoked by using the system call **syscall()** which takes the system call table entry number and arguments as parameters and then traps to the kernel.

To generate a system call stub so that an ordinary C function call will invoke the new system call, you also need to edit the **include/asm/unistd.h** file, as follows, so that it can be used.

```c
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
...  
#define __NR_sched_get_priority_min 160  
#define __NR_sched_rr_get_interval 161  
#define __NR_nanosleep 162  
#define __NR_mremap 163  
#define __NR_poll 168  
#define __NR_getpmsg 188  
#define __NR_putpmsg 189  
#define __NR_vfork 190  
#define __NR_my_new_call 191
```

Finally, you need to generate the system call stub. These constant definitions are used to create the system call stub function for use with C programs for no arguments, one argument, and so on.

**Generating a System Call Stub**

The system call stub is generated by using a macro call from a user-space program. Macros are available for generating a stub with zero to five parameters. For example, the macro for generating a stub with two arguments has the form

```c
_syscall2(type, name, type1, arg1, type2, arg2);
```

In this macro, **type** is the type of the return value of the system call stub, name is the name of the stub, **type1** is the type of the first argument, **arg1**, and **type2** is the type of the second argument, **arg2**. These macros are defined in **include/linux/unistd.h** (which includes the file **include/asm/unistd.h**).

You can generate the stub by making the following macro call in your user program.

```c
#include <linux/unistd.h>
/* Generate system call for int foo(char *baz, double bar) */
_syscall2(int, foo, char *, baz, double, bar);
```
Also, a system function, `system_call()`, defined in `arch/i386/entry.S`, can be used to invoke a kernel function (without generating a stub). For example, if the index in the `sys_call_table` for `foo()` is 193, then you can call the imaginary `foo()` function with the following.

```c
#include <sys/syscall.h>
syscall(193, &baz_arg, bar_arg);
```

**Kernel Function Organization**

A kernel function is an ordinary C function compiled to execute in supervisor mode with the rest of the kernel. Other than its header, it requires no particular organization, since it can perform any task that its author chooses.

Consider the simplest kind of kernel function, one that performs some action without accepting a parameter or returning a value (this function is hypothetical, since the kernel contains no such functions).

```c
asmlinkage void sys_foo(void) {
    /* Write a value to the console */
}
```

Suppose that `sys_foo()` is a real kernel function (it is not). If a user program calls it by using `system_call()`, then `sys_call_table[NR_foo]` will have an entry and will contain the entry point address for `sys_foo()` (see handout). `NR_foo` will be set in `include/asm/unistd.h`, and the `sys_call_table` entry will be set to the address of `sys_foo()` in `arch/i386/kernel/entry.S`. If a stub had been created, then when a user program called `void foo(void)` the kernel would begin executing at the entry point for `asmlinkage void sys_foo(void)`. After the function finished, the kernel would return to the user program (via the `ret_from_sys_call` sequence).

A slightly more complex function, such as `sys_getuid()`, returns a value, as follows.

```c
asmlinkage int sys_getuid(void) {
    return current->uid;
}
```

In this case, the `uid_t getuid()` stub traps to `sys_call_table[NR_getuid]` (that is, to `sys_call_table[24]`, which points to the entry point for `asmlinkage int sys_getuid(void)`). The `current` variable is global to the kernel and references the `struct task_struct` of the currently executing process, so `current->uid` is the user id for the current process. When the `sys_getuid()` function returns, its return value is placed on the user-space stack so that the user process can receive the result.

Now consider a kernel function that takes one or more arguments, such as this one.

```c
asmlinkage int sys_close(unsigned int fd) {
    int error;
    struct file *filp;
    struct files_struct *files;
    files = current->files;
    error = -EBADF;
    if(fd < NR_OPEN && (filp = files->fd[fd]) != NULL) {
        ...
    }
    return error;
}
```

The `fd` input parameter is passed to `sys_close()` by the `system_call()` function, simply by passing the argument value that it received on the stack when the system call stub was called. As
mentioned in a previous section, kernel functions are called indirectly (via the `sys_call_table`), so the C compiler does not check the type and number of actual parameters. Therefore it is prudent for the kernel function to perform runtime checks on the values that are passed to determine whether they are reasonable. (`sys_close()` does not check `fd` before using it). Thus if it gets a bad parameter, then it will attempt to reference an element of the `files->fd` array. However, it passes `fd` on to other routines that can check it before real harm is done.) If you write a kernel function and do not check the parameters prior to using them, you might crash the system.

**Referencing User-Space Memory Locations**

Your function might have a call-by-reference argument in which the kernel function needs to write information into a user-space address. That is, the argument is a pointer to a variable that was declared in the calling function. Kernel functions execute in the kernel data segment (see handout), so the memory translation mechanism does not allow a process that is executing in kernel space to write into a user segment without changing the state of the protection mechanism. To read or write user-space memory from the kernel, the kernel first should check to see whether the address is legitimately defined in the user virtual address space by using the following function:

```
verify_area(int type, const void *addr, unsigned long size);
```

This function checks the validity of a read operation (type is set to `VERIFY_READ`) or a write operation (type is set to `VERIFY_WRITE`). The addr argument specifies the address to be verified, and the size argument is the number of bytes in the block of memory to be verified. `verify_area()` returns zero if the operation is permitted on the memory area, and nonzero otherwise. Following is a typical code fragment to verify the kernel's ability to read a memory block named `buf` of length `buf_len`.

```
flag = verify_area(VERIFY_READ, buf, buf_len);
if(flag) {
    // Error—unable to read buf
}
```

If the block of memory can be read or written by the kernel, then the following two functions are used to actually read and write the user address space:

```
memcpy_fromfs(void *to, void *from, unsigned long n);
memcpy_tofs(void *to, void *from, unsigned long n);
```

`memcpy_fromfs()` is used to read the user-space address, and `memcpy_tofs()` is used to write the user-space address. The `to` argument is the destination of the data copy operation, and the `from` argument is the source. The `n` argument is the number of bytes to be copied.

As a better (newer) alternative, you can make use of `copy_to_user()` and `copy_from_user()` calls which realize the same function. Usage of these calls is summarized as

```
#include <asm/uaccess.h>
er = get_user(x, addr);
er = put_user(x, addr);
bytes_left = copy_from_user(void *to, const void *from, unsigned long n);
bytes_left = copy_to_user(void *to, const void *from, unsigned long n);
```

Above two macros (`put_user` and `get_user`) transfer data between kernel space and user space. In the first example, the kernel variable `x` gets the value of the thing pointed to by `addr` (in user space). For `put_user`, the value of the variable `x` is written starting at the address `addr`. Note well that `x` is a variable, not a pointer. `addr` must be a properly typed pointer, since its type determines
number of bytes that are copied. More generically, copy_from_user copies \( n \) bytes from address from in user space to address to in kernel space. copy_to_user copies in the opposite direction. None of these calls need the (good, old) verify_area() call, as they do all the area verification on their own using the paging unit in the CPU hardware. (Introducing less chance of missing address verification). Also, the new address verification is much faster than the old scheme was. get_user and put_user return 0 for success and -EFAULT for bad access. copy_from_user and copy_to_user return the number of bytes they failed to copy (so again zero is a successful return).

**Problem Statement**

**Part A**

Design and implement a new system call, modtimeCheck(), that takes a filename and returns its modification time. It should return zero on success and -1 if an error occurs. Here is how the call look like:

```c
int modtimeCheck(char *filename, time_t *mod_time)
```

Actually, this function is not a new discovery, and (its superset) has been implemented to be available in stat() system call (see `man 2 stat`). All you have to do is to trace down the stat() system call (starting from system call table) to see the place it retrieves the file’s last modification time.

You are advised to take a conservative, incremental strategy for developing this system call (and any kernel work). First focus on putting printk() statements into some of the key parts of the Linux you interfere with, in order to build up confidence as to where to add your modifications.

**Part B**

Write a user-space program to test modtimeCheck(). Modify the file you observe within your test code to make sure that your system call works perfect. The program should also create a stub for your new system call function, so that it may be called directly with its name.

**Attacking the Problem**

**The Kernel printk() Function**

When writing kernel code, you often will want to print messages to stdout as you develop and debug it. (Note that if your program is using printk(), then the process that executes the code must be running with root's user id.) Of course, software that implements the kernel does not have thestdio library available to it; thus you cannot necessarily use printf() to write to stdout. Kernel programmers decided many years ago that they could not live without a print statement and also that they did not want to rely on printf()'s working in the kernel, so they developed their own kernel version of printf(), called printk(). printk() behaves the same as printf(); it actually is implemented by using printf() in Linux (see `/usr/linux/include/linux/kernel.h`). You may check the output of printk() function by looking at the kernel logger daemon output via dmesg command.

**Organizing a Solution**

To come up with the solution, make sure you understand how stat() retrieves the necessary values. After that, it should be very easy to complete this exercise on the Linux kernel. You need to learn many small details in order to get your first kernel function to work properly. So you are advised to use a conservative, incremental strategy for developing your first kernel function. Here are some guidelines.
For your first debug version, focus on getting the system call interface to work properly. It should not take any arguments or return any values. Instead, simply do a printk() within the function body so that you can see that you have successfully implemented a complete function in the kernel and that the system call interface is working properly.

Next, create a dummy version that passes an argument into the kernel (call-by-value) but that does not expect the kernel to write anything back into the user space.

Your third version should be a simple call-by-reference call. You will be using verify_area() to reference user space, memcpy_fromfs() to read information from the user space, and memcpy_tofs() to write data when it is returned by reference (Or get_user(), put_user() as explained earlier).

After you successfully complete the above steps, you can then proceed to implement the code for the functionality explained in the problem statement. Here is a possible code skeleton for your kernel function.

```c
asmlinkage int sys_modtimeCheck(char *filename, time_t *mod_time) {
/* You will be conservative and block interrupts while retrieving data */
    cli(); /* Disable interrupts */
    ... /* Get data */
    sti(); /* Enable interrupts */

    return 0;
}
```

After you complete your implementation of sys_modtimeCheck(), you may put it in fs/stat.c. Be sure to save the original so that you can restore it after you have completed this exercise. Alternatively, you can put the new function in your own new file. In this case, you will need to edit the Makefile in the kernel directory in which the new file is placed. Add the name of the new file to the O_OBJS list in the Makefile.

**Rebuilding the Kernel**

Ultimately, the kernel is just another program that must be compiled and then linked into its runtime environment in preparation for execution. Unlike other programs, however, its environment is determined by the logical configuration—the configuration dictated by the hardware and the system administrator’s parameter choices. Thus it cannot depend on the existence of other software in its runtime environment.

Software engineers have spent many years refining the technique for building UNIX. Linux uses those accumulated tools in its build environment. Because you are using a running Linux machine that has the source code, the build environment should already be in place on the machine (it is installed with the source code).

The build procedure takes place from the base directory of the Linux source, typically in /usr/src/linux, though your linux source directory might be installed elsewhere. /usr/src/linux is the source subtree described in lab hour The linux directory contains the directories that have source code, include files, and library routines. It also includes the files README and Makefile. You should read the README file, even if you cannot understand all of it at this point. It provides information regarding, for example, where the build environment should be installed (in /usr/src) and what to do if things break badly while you are doing kernel work.

The Makefile in the linux root directory is the top-level makefile for building the Linux kernel. As you might expect, it is relatively complex, as it is designed to automate most of the details of
building and installing a new kernel. It does this by invoking various other tools and by using other makefiles in subdirectories. In a properly installed build environment, the kernel is rebuilt with only five commands (executed by a superuser):

```
# make clean
# make <config_opt>
# make depend
# make
# make <boot_opt>
```

All of these commands use `linux/Makefile` to create a new kernel and store it in `linux/arch/i386/boot/bzImage`.

Next, you will examine each command more closely.

- **make clean**: Removes old relocatable object files and other temporary files so that your build will have a clean environment in which it can be built.

- **make <config_opt>**: Defines the logical environment in which the new kernel should exist. Generally, the `<config_opt>` parameter can be set to `config`, `menuconfig` or `xconfig`. (See Linux Kernel HOWTO on the course web page for more details) In any case, make will run an interface, which builds a configuration file, `linux/arch/i386/config.in`. You usually will not want to change the `config.in` file, though you should inspect it to get an idea of how it works. (If you decide to change the config.in file, you will need to dig deeper into the tools and configurations than you will in this manual. To use the existing config.in, as well as the default options for making the kernel configuration, use the `oldconfig` value for `<config_opt>` rather than the config value for this step. (or equivalently, load the properly saved configuration file from `configs` directory) This part of Makefile creates `linux/.config` and `linux/include/linux/autoconf.h` files. The `.config` file captures the responses from the interactive dialog that determines the exact configuration; if you run `make config` again, it will use the responses from `.config`. The `autoconf.h` file is a header file that is used to invoke compile-time options that correspond to the choices made in the interactive dialog with make.

- **make depend**: Many files must be compiled and in a particular order. make depend creates a file, `linux/.depend`, that specifies the compile order dependencies. Specifying dependencies is a simple, but laborious, task that is automated in the depend option of `Makefile`.

- **make**: Compiles all of the kernel source code, producing `linux/vmlinux`, the kernel executable file. If you have written new kernel code, or modified existing files, then this step will compile that code (therefore this is probably where you will encounter your first problems in building a new kernel). This is the most complex part of `Makefile`. It invokes `make` on all of the subdirectories and then ultimately links the results together to form the kernel executable.

- **make <boot_opt>**: Compresses the `linux/vmlinux` file to create a bootable kernel image and installs it in `linux/arch/i386/boot/bzImage`. This can be done with `<boot_opt>` set to `bzImage`. To make a copy of the bootable kernel image on a floppy disk, use `zdisk` instead of `bzimage` as the `<boot_opt>` parameter. In Version 2.2.x, a kernel that you generate might be too large. If you set `<boot_opt>` to `bzdisk`, you should be able to generate a boot floppy disk.

Performing the steps to build a new kernel is relatively easy because the detailed work has been encapsulated in the `linux/Makefile`, the subdirectory makefiles, and the configuration files. Before you attempt to create a new bootable kernel, be sure to save `bzImage` so that you can restore it when needed. See the `README` file in the `linux` directory for more information.
Submission
Your group will be presenting your solution on the due date announced on the course web page. Watch for announcements on the web page for up-to-date-details. You are supposed to obey the academic honesty rules posted to the web page of the course. Discarding them may cause you trouble. You may send your questions regarding this text to course e-mail group, which is cse331@cse.yeditepe.edu.tr.
Have a nice project!

Happy codings,

Okan Türkeş