Project—Adding a System Call to the Linux Kernel

In this project, you will study the system call interface provided by the Linux operating system and how user programs communicate with the operating system kernel via this interface. Your task is to incorporate a new system call into the kernel, thereby expanding the functionality of the operating system.

Getting Started

A user-mode procedure call is performed by passing arguments to the called procedure either on the stack or through registers, saving the current state and the value of the program counter, and jumping to the beginning of the code corresponding to the called procedure. The process continues to have the same privileges as before.

System calls appear as procedure calls to user programs, but result in a change in execution context and privileges. In Linux on the Intel 386 architecture, a system call is accomplished by storing the system call number into the EAX register, storing arguments to the system call in other hardware registers, and executing a trap instruction (which is the INT 0x80 assembly instruction). After the trap is executed, the system call number is used to index into a table of code pointers to obtain the starting address for the handler code implementing the system call. The process then jumps to this address and the privileges of the process are switched from user to kernel mode. With the expanded privileges, the process can now execute kernel code that might

include privileged instructions that cannot be executed in user mode. The kernel code can then perform the requested services such as interacting with I/O devices, perform process management and other such activities that cannot be performed in user mode.

The system call numbers for recent versions of the Linux kernel are listed in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (For instance, `__NR_close`, which corresponds to the system call `close()` that is invoked for closing a file descriptor, is defined as value 6.) The list of pointers to system call handlers is typically stored in the file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` under the heading `ENTRY(sys_call_table)`. Notice that `sys_close` is stored at entry numbered 6 in the table to be consistent with the system call number defined in `unistd.h` file. (The keyword `.long` denotes that the entry will occupy the same number of bytes as a data value of type `long`.)

Building a New Kernel

Before adding a system call to the kernel, you must familiarize yourself with the task of building the binary for a kernel from its source code and booting the machine with the newly built kernel. This activity comprises the following tasks, some of which are dependent on the particular installation of the Linux operating system.

- Obtain the kernel source code for the Linux distribution. If the source code package has been previously installed on your machine, the corresponding files might be available under `/usr/src/linux` or `/usr/src/linux-2.x` (where the suffix corresponds to the kernel version number). If the package has not been installed earlier, it can be downloaded from the provider of your Linux distribution or from `http://www.kernel.org`.
- Learn how to configure, compile, and install the kernel binary. This will vary between the different kernel distributions, but some typical commands for building the kernel (after entering the directory where the kernel source code is stored) include:
  - `make xconfig`
  - `make dep`
  - `make bzImage`
- Add a new entry to the set of bootable kernels supported by the system. The Linux operating system typically uses utilities such as `lilo` and `grub` to maintain a list of bootable kernels, from which the user can choose during machine boot-up. If your system supports `lilo`, add an entry to `lilo.conf`, such as:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

where `/boot/bzImage.mykernel` is the kernel image and `mykernel` is

the label associated with the new kernel allowing you to choose it during bootup process. By performing this step, you have the option of either booting a new kernel or booting the unmodified kernel if the newly built kernel does not function properly.

Extending Kernel Source

You can now experiment with adding a new file to the set of source files used for compiling the kernel. Typically, the source code is stored in the `/usr/src/linux-2.x/kernel` directory, although that location may differ in your Linux distribution. There are two options for adding the system call. The first is to add the system call to an existing source file in this directory. A second option is to create a new file in the source directory and modify `/usr/src/linux-2.x/kernel/Makefile` to include the newly created file in the compilation process. The advantage of the first approach is that by modifying an existing file that is already part of the compilation process, the Makefile does not require modification.

Adding a System Call to the Kernel

Now that you are familiar with the various background tasks corresponding to building and booting Linux kernels, you can begin the process of adding a new system call to the Linux kernel. In this project, the system call will have limited functionality; it will simply transition from user mode to kernel mode, print a message that is logged with the kernel messages, and transition back to user mode. We will call this the *helloworld* system call. While it has only limited functionality, it illustrates the system call mechanism and sheds light on the interaction between user programs and the kernel.

- Create a new file called `helloworld.c` to define your system call. Include the header files `linux/linkage.h` and `linux/kernel.h`. Add the following code to this file:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
   printk(KERN_EMERG "hello world!");

   return 1;
}
```

  This creates a system call with the name `sys_helloworld()`. If you choose to add this system call to an existing file in the source directory, all that is necessary is to add the `sys_helloworld()` function to the file you choose. `asmlinkage` is a remnant from the days when Linux used both C++ and C code and is used to indicate that the code is written in C. The `printk()` function is used to print messages to a kernel log file and therefore may only be called from the kernel. The kernel messages specified in the parameter to `printk()` are logged in the file `/var/log/kernel/warnings`. The function prototype for the `printk()` call is defined in `/usr/include/linux/kernel.h`.

- Define a new system call number for `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. A user program can use this number to identify the newly added system call. Also be sure to increment the value for `__NR_syscalls`, which is also stored in the same file. This constant tracks the number of system calls currently defined in the kernel.
- Add an entry `.long sys_helloworld` to the `sys_call_table` defined in `/usr/src/linux-2.x/arch/i386/kernel/entry.S` file. As discussed earlier, the system call number is used to index into this table to find the position of the handler code for the invoked system call.
- Add your file `helloworld.c` to the Makefile (if you created a new file for your system call.) Save a copy of your old kernel binary image (in case there are problems with your newly created kernel.) You can now build the new kernel, rename it to distinguish it from the unmodified kernel, and add an entry to the loader configuration files (such as `lilo.conf`). After completing these steps, you may now boot either the old kernel or the new kernel that contains your system call inside it.

Using the System Call From a User Program

When you boot with the new kernel it will support the newly defined system call; it is now simply a matter of invoking this system call from a user program. Ordinarily, the standard C library supports an interface for system calls defined for the Linux operating system. As your new system call is not linked into the standard C library, invoking your system call will require manual intervention.

As noted earlier, a system call is invoked by storing the appropriate value into a hardware register and performing a trap instruction. Unfortunately, these are low-level operations that cannot be performed using C language statements and instead require assembly instructions. Fortunately, Linux provides macros for instantiating wrapper functions that contain the appropriate assembly instructions. For instance, the following C program uses the `_syscall0()` macro to invoke the newly defined system call:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
   helloworld();
}
```

- The `_syscall0` macro takes two arguments. The first specifies the type of the value returned by the system call; the second argument is the name of the system call. The name is used to identify the system call number that is stored in the hardware register before the trap instruction is executed.

If your system call requires arguments, then a different macro (such as `_syscall0`, where the suffix indicates the number of arguments) could be used to instantiate the assembly code required for performing the system call.

- Compile and execute the program with the newly built kernel. There should be a message "`hello world!`" in the kernel log file `/var/log/kernel/warnings` to indicate that the system call has executed.

As a next step, consider expanding the functionality of your system call. How would you pass an integer value or a character string to the system call and have it be printed into the kernel log file? What are the implications for passing pointers to data stored in the user program's address space as opposed to simply passing an integer value from the user program to the kernel using hardware registers?