**Chapter 8. System Calls**

**8.1. System Calls**

So far, the only thing we've done was to use well defined kernel mechanisms to register /proc files and device handlers. This is fine if you want to do something the kernel programmers thought you'd want, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you're mostly on your own.

This is where kernel programming gets dangerous. While writing the example below, I killed the open() system call. This meant I couldn't open any files, I couldn't run any programs, and I couldn't shutdown the computer. I had to pull the power switch. Luckily, no files died. To ensure you won't lose any files either, please run sync right before you do the insmod and the rmmod.

Forget about /proc files, forget about device files. They're just minor details. The real process to kernel communication mechanism, the one used by all processes, is system calls. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism used. If you want to change the behaviour of the kernel in interesting ways, this is the place to do it. By the way, if you want to see which system calls a program uses, run strace <arguments>. In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that's the reason why it's called `protected mode').

System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no

longer running in restricted user mode, but as the operating system kernel −−− and therefore you're allowed to do whatever you want.

The location in the kernel a process can jump to is called system\_call. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (sys\_call\_table) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it's at the source file arch/$<$architecture$>$/kernel/entry.S, after the line ENTRY(system\_call).

So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at sys\_call\_table to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it's important for cleanup\_module to restore the table to its original state.

The source code here is an example of such a kernel module. We want to `spy' on a certain user, and to printk() a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called our\_sys\_open. This function checks the uid (user's id) of the current process, and if it's equal to the uid we spy on, it calls printk() to display the name of the file to be opened. Then, either way, it calls the original open() function with the same parameters, to actually open the file.

The init\_module function replaces the appropriate location in sys\_call\_table and keeps the original pointer in a variable. The cleanup\_module function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's open system call will be A\_open and B's will be B\_open. Now, when A is inserted into the kernel, the system call is replaced with A\_open, which will call the original sys\_open when it's done. Next, B is inserted into the kernel, which replaces the system call with B\_open, which will call what it thinks is the original system call, A\_open, when it's done.

Now, if B is removed first, everything will be well−−−it will simply restore the system call to A\_open, which calls the original. However, if A is removed and then B is removed, the system will crash. A's removal will restore the system call to the original, sys\_open, cutting B out of the loop. Then, when B is removed, it will restore the system call to what it thinks is the original, A\_open, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won't change the system call when it's removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to B\_open so that it is no longer pointing to A\_open, so it won't restore it to sys\_open before it is removed from memory.

Unfortunately, B\_open will still try to call A\_open which is no longer there, so that even without removing B the system would crash. Note that all the related problems make syscall stealing unfeasiable for production use. In order to keep people from doing potential harmful things sys\_call\_table is no longer exported. This means, if you want to do something more than a mere dry run of this example, you will have to patch your current kernel in order to have sys\_call\_table exported. In the example directory you will find a README and the patch. As you can imagine, such modifications are not to be taken lightly. Do not try this on valueable systems (ie systems that you do not own − or cannot restore easily). You'll need to get the complete sourcecode of this guide as a tarball in order to get the patch and the README. Depending on your kernel version, you might even need to

hand apply the patch. Still here? Well, so is this chapter. If Wyle E. Coyote was a kernel hacker, this would be the first thing he'd try. ;)

Example 8−1. syscall.c

/\*

\* syscall.c

\*

\* System call "stealing" sample.

\*/

/\*

\* Copyright (C) 2001 by Peter Jay Salzman

\*/

/\*

\* The necessary header files

\*/

/\*

\* Standard in kernel modules

\*/

#include <linux/kernel.h> /\* We're doing kernel work \*/

#include <linux/module.h> /\* Specifically, a module, \*/

#include <linux/moduleparam.h> /\* which will have params \*/

#include <linux/unistd.h> /\* The list of system calls \*/

/\*

\* For the current (process) structure, we need

\* this to know who the current user is.

\*/

#include <linux/sched.h>

#include <asm/uaccess.h>

/\*

\* The system call table (a table of functions). We

\* just define this as external, and the kernel will

\* fill it up for us when we are insmod'ed

\*

\* sys\_call\_table is no longer exported in 2.6.x kernels.

\* If you really want to try this DANGEROUS module you will

\* have to apply the supplied patch against your current kernel

\* and recompile it.

\*/

extern void \*sys\_call\_table[];

/\*

\* UID we want to spy on − will be filled from the

\* command line

\*/

static int uid;

module\_param(uid, int, 0644);

/\*

\* A pointer to the original system call. The reason

\* we keep this, rather than call the original function

\* (sys\_open), is because somebody else might have

\* replaced the system call before us. Note that this

\* is not 100% safe, because if another module

\* replaced sys\_open before us, then when we're inserted

\* we'll call the function in that module − and it

\* might be removed before we are.

\*

\* Another reason for this is that we can't get sys\_open.

\* It's a static variable, so it is not exported.

\*/

asmlinkage int (\*original\_call) (const char \*, int, int);

/\*

\* The function we'll replace sys\_open (the function

\* called when you call the open system call) with. To

\* find the exact prototype, with the number and type

\* of arguments, we find the original function first

\* (it's at fs/open.c).

\*

\* In theory, this means that we're tied to the

\* current version of the kernel. In practice, the

\* system calls almost never change (it would wreck havoc

\* and require programs to be recompiled, since the system

\* calls are the interface between the kernel and the

\* processes).

\*/

asmlinkage int our\_sys\_open(const char \*filename, int flags, int mode)

{

int i = 0;

char ch;

/\*

\* Check if this is the user we're spying on

\*/

The Linux Kernel Module Programming Guide

Chapter 8. System Calls 52

if (uid == current−>uid) {

/\*

\* Report the file, if relevant

\*/

printk("Opened file by %d: ", uid);

do {

get\_user(ch, filename + i);

i++;

printk("%c", ch);

} while (ch != 0);

printk("\n");

}

/\*

\* Call the original sys\_open − otherwise, we lose

\* the ability to open files

\*/

return original\_call(filename, flags, mode);

}

/\*

\* Initialize the module − replace the system call

\*/

int init\_module()

{

/\*

\* Warning − too late for it now, but maybe for

\* next time...

\*/

printk(KERN\_ALERT "I'm dangerous. I hope you did a ");

printk(KERN\_ALERT "sync before you insmod'ed me.\n");

printk(KERN\_ALERT "My counterpart, cleanup\_module(), is even");

printk(KERN\_ALERT "more dangerous. If\n");

printk(KERN\_ALERT "you value your file system, it will ");

printk(KERN\_ALERT "be \"sync; rmmod\" \n");

printk(KERN\_ALERT "when you remove this module.\n");

/\*

\* Keep a pointer to the original function in

\* original\_call, and then replace the system call

\* in the system call table with our\_sys\_open

\*/

original\_call = sys\_call\_table[\_\_NR\_open];

sys\_call\_table[\_\_NR\_open] = our\_sys\_open;

/\*

\* To get the address of the function for system

\* call foo, go to sys\_call\_table[\_\_NR\_foo].

\*/

printk(KERN\_INFO "Spying on UID:%d\n", uid);

return 0;

}

/\*

\* Cleanup − unregister the appropriate file from /proc

\*/

void cleanup\_module()

{

/\*

\* Return the system call back to normal

\*/

if (sys\_call\_table[\_\_NR\_open] != our\_sys\_open) {

printk(KERN\_ALERT "Somebody else also played with the ");

printk(KERN\_ALERT "open system call\n");

printk(KERN\_ALERT "The system may be left in ");

printk(KERN\_ALERT "an unstable state.\n");

}

sys\_call\_table[\_\_NR\_open] = original\_call;

}

Chapter 9. Blocking Processes

9.1. Blocking Processes

What do you do when somebody asks you for something you can't do right away? If you're a human being and you're bothered by a human being, the only thing you can say is: "Not right now, I'm busy. Go away!". But if you're a kernel module and you're bothered by a process, you have another possibility. You can put the process to sleep until you can service it. After all, processes are being put to sleep by the kernel and woken up all the time (that's the way multiple processes appear to run on the same time on a single CPU).

This kernel module is an example of this. The file (called /proc/sleep) can only be opened by a single process at a time. If the file is already open, the kernel module calls wait\_event\_interruptible[12]. This function changes the status of the task (a task is the kernel data structure which holds information about a process and the system call it's in, if any) to TASK\_INTERRUPTIBLE, which means that the task will not run until it is woken up somehow, and adds it to WaitQ, the queue of tasks waiting to access the file. Then, the function calls the scheduler to context switch to a different process, one which has some use for the CPU.

When a process is done with the file, it closes it, and module\_close is called. That function wakes up all the processes in the queue (there's no mechanism to only wake up one of them). It then returns and the process which just closed the file can continue to run. In time, the scheduler decides that that process has had enough and gives control of the CPU to another process. Eventually, one of the processes which was in the queue will be given control of the CPU by the scheduler. It starts at the point right after the call to module\_interruptible\_sleep\_on[13]. It can then proceed to set a global variable to tell all the other processes that the file is still open and go on with its life. When the other processes get a piece of the CPU,

they'll see that global variable and go back to sleep.

So we'll use tail −f to keep the file open in the background, while trying to access it with another process (again in the background, so that we need not switch to a different vt). As soon as the first background process is killed with kill %1 , the second is woken up, is able to access the file and finally terminates.

To make our life more interesting, module\_close doesn't have a monopoly on waking up the processes which wait to access the file. A signal, such as Ctrl+c (SIGINT) can also wake up a process. [14] In that case, we want to return with −EINTR immediately. This is important so users can, for example, kill the process before it receives the file.

There is one more point to remember. Some times processes don't want to sleep, they want either to get what they want immediately, or to be told it cannot be done. Such processes use the O\_NONBLOCK flag when opening the file. The kernel is supposed to respond by returning with the error code −EAGAIN from operations which would otherwise block, such as opening the file in this example. The program cat\_noblock, available in the source directory for this chapter, can be used to open a file with O\_NONBLOCK.

hostname:~/lkmpg−examples/09−BlockingProcesses# insmod sleep.ko

hostname:~/lkmpg−examples/09−BlockingProcesses# cat\_noblock /proc/sleep

Last input:

hostname:~/lkmpg−examples/09−BlockingProcesses# tail −f /proc/sleep &

Last input:

Last input:

Last input:

Last input:

Last input:

Last input:

Last input:

tail: /proc/sleep: file truncated

[1] 6540

hostname:~/lkmpg−examples/09−BlockingProcesses# cat\_noblock /proc/sleep

Open would block

hostname:~/lkmpg−examples/09−BlockingProcesses# kill %1

[1]+ Terminated tail −f /proc/sleep

hostname:~/lkmpg−examples/09−BlockingProcesses# cat\_noblock /proc/sleep

Last input:

hostname:~/lkmpg−examples/09−BlockingProcesses#

Example 9−1. sleep.c

/\*

\* sleep.c − create a /proc file, and if several processes try to open it at

\* the same time, put all but one to sleep

\*/

#include <linux/kernel.h> /\* We're doing kernel work \*/

#include <linux/module.h> /\* Specifically, a module \*/

#include <linux/proc\_fs.h> /\* Necessary because we use proc fs \*/

#include <linux/sched.h> /\* For putting processes to sleep and

waking them up \*/

#include <asm/uaccess.h> /\* for get\_user and put\_user \*/

/\*

\* The module's file functions

\*/

/\*

\* Here we keep the last message received, to prove that we can process our

\* input

\*/

#define MESSAGE\_LENGTH 80

static char Message[MESSAGE\_LENGTH];

static struct proc\_dir\_entry \*Our\_Proc\_File;

#define PROC\_ENTRY\_FILENAME "sleep"

/\*

\* Since we use the file operations struct, we can't use the special proc

\* output provisions − we have to use a standard read function, which is this

\* function

\*/

static ssize\_t module\_output(struct file \*file, /\* see include/linux/fs.h \*/

char \*buf, /\* The buffer to put data to(in the user segment) \*/

size\_t len, /\* The length of the buffer \*/ loff\_t \* offset)

{

static int finished = 0;

int i;

char message[MESSAGE\_LENGTH + 30];

/\*

\* Return 0 to signify end of file − that we have nothing

\* more to say at this point.

\*/

if (finished) {

finished = 0;

return 0;

}

/\*

\* If you don't understand this by now, you're hopeless as a kernel

\* programmer.

\*/

sprintf(message, "Last input:%s\n", Message);

for (i = 0; i < len && message[i]; i++)

put\_user(message[i], buf + i);

finished = 1;

return i; /\* Return the number of bytes "read" \*/

}

/\*

\* This function receives input from the user when the user writes to the /proc

\* file.

\*/

static ssize\_t module\_input(struct file \*file, /\* The file itself \*/

const char \*buf, /\* The buffer with input \*/ size\_t length, /\* The buffer's length \*/ loff\_t \* offset)

{ /\* offset to file − ignore \*/

int i;

/\*

\* Put the input into Message, where module\_output will later be

\* able to use it

\*/

for (i = 0; i < MESSAGE\_LENGTH − 1 && i < length; i++)

get\_user(Message[i], buf + i);

/\*

\* we want a standard, zero terminated string

\*/

Message[i] = '\0';

/\*

\* We need to return the number of input characters used

\*/

return i;

}

/\*

\* 1 if the file is currently open by somebody

\*/

int Already\_Open = 0;

/\*

\* Queue of processes who want our file

\*/

DECLARE\_WAIT\_QUEUE\_HEAD(WaitQ);

/\*

\* Called when the /proc file is opened

\*/

static int module\_open(struct inode \*inode, struct file \*file)

{

/\*

\* If the file's flags include O\_NONBLOCK, it means the process doesn't

\* want to wait for the file. In this case, if the file is already

\* open, we should fail with −EAGAIN, meaning "you'll have to try

\* again", instead of blocking a process which would rather stay awake.

\*/

if ((file−>f\_flags & O\_NONBLOCK) && Already\_Open)

return −EAGAIN;

/\*

\* This is the correct place for try\_module\_get(THIS\_MODULE) because

\* if a process is in the loop, which is within the kernel module,

\* the kernel module must not be removed.

\*/

try\_module\_get(THIS\_MODULE);

/\*

\* If the file is already open, wait until it isn't

\*/

while (Already\_Open) {

int i, is\_sig = 0;

/\*

\* This function puts the current process, including any system

\* calls, such as us, to sleep. Execution will be resumed right

\* after the function call, either because somebody called

\* wake\_up(&WaitQ) (only module\_close does that, when the file

\* is closed) or when a signal, such as Ctrl−C, is sent

\* to the process

\*/

wait\_event\_interruptible(WaitQ, !Already\_Open);

/\*

\* If we woke up because we got a signal we're not blocking,

\* return −EINTR (fail the system call). This allows processes

\* to be killed or stopped.

\*/

/\*

\* Emmanuel Papirakis:

\*

\* This is a little update to work with 2.2.\*. Signals now are contained in

\* two words (64 bits) and are stored in a structure that contains an array of

\* two unsigned longs. We now have to make 2 checks in our if.

\*

\* Ori Pomerantz:

\*

\* Nobody promised me they'll never use more than 64 bits, or that this book

\* won't be used for a version of Linux with a word size of 16 bits. This code

\* would work in any case.

\*/

for (i = 0; i < \_NSIG\_WORDS && !is\_sig; i++)

is\_sig =

current−>pending.signal.sig[i] & ~current−>

blocked.sig[i];

if (is\_sig) {

/\*

\* It's important to put module\_put(THIS\_MODULE) here,

\* because for processes where the open is interrupted

\* there will never be a corresponding close. If we

\* don't decrement the usage count here, we will be

\* left with a positive usage count which we'll have no

\* way to bring down to zero, giving us an immortal

\* module, which can only be killed by rebooting

\* the machine.

\*/

The Linux Kernel Module Programming Guide

Chapter 9. Blocking Processes 58

module\_put(THIS\_MODULE);

return −EINTR;

}

}

/\*

\* If we got here, Already\_Open must be zero

\*/

/\*

\* Open the file

\*/

Already\_Open = 1;

return 0; /\* Allow the access \*/

}

/\*

\* Called when the /proc file is closed

\*/

int module\_close(struct inode \*inode, struct file \*file)

{

/\*

\* Set Already\_Open to zero, so one of the processes in the WaitQ will

\* be able to set Already\_Open back to one and to open the file. All

\* the other processes will be called when Already\_Open is back to one,

\* so they'll go back to sleep.

\*/

Already\_Open = 0;

/\*

\* Wake up all the processes in WaitQ, so if anybody is waiting for the

\* file, they can have it.

\*/

wake\_up(&WaitQ);

module\_put(THIS\_MODULE);

return 0; /\* success \*/

}

/\*

\* This function decides whether to allow an operation (return zero) or not

\* allow it (return a non−zero which indicates why it is not allowed).

\*

\* The operation can be one of the following values:

\* 0 − Execute (run the "file" − meaningless in our case)

\* 2 − Write (input to the kernel module)

\* 4 − Read (output from the kernel module)

\*

\* This is the real function that checks file permissions. The permissions

\* returned by ls −l are for reference only, and can be overridden here.

\*/

static int module\_permission(struct inode \*inode, int op, struct nameidata \*nd)

{

/\*

\* We allow everybody to read from our module, but only root (uid 0)

\* may write to it

\*/

if (op == 4 || (op == 2 && current−>euid == 0))

return 0;

/\*

\* If it's anything else, access is denied

\*/

return −EACCES;

}

/\*

\* Structures to register as the /proc file, with pointers to all the relevant

\* functions.

\*/

/\*

\* File operations for our proc file. This is where we place pointers to all

\* the functions called when somebody tries to do something to our file. NULL

\* means we don't want to deal with something.

\*/

static struct file\_operations File\_Ops\_4\_Our\_Proc\_File = {

.read = module\_output, /\* "read" from the file \*/

.write = module\_input, /\* "write" to the file \*/

.open = module\_open, /\* called when the /proc file is opened \*/

.release = module\_close, /\* called when it's closed \*/

};

/\*

\* Inode operations for our proc file. We need it so we'll have somewhere to

\* specify the file operations structure we want to use, and the function we

\* use for permissions. It's also possible to specify functions to be called

\* for anything else which could be done to an inode (although we don't bother,

\* we just put NULL).

\*/

static struct inode\_operations Inode\_Ops\_4\_Our\_Proc\_File = {

.permission = module\_permission, /\* check for permissions \*/

};

/\*

\* Module initialization and cleanup

\*/

/\*

\* Initialize the module − register the proc file

\*/

int init\_module()

{

Our\_Proc\_File = create\_proc\_entry(PROC\_ENTRY\_FILENAME, 0644, NULL);

if (Our\_Proc\_File == NULL) {

remove\_proc\_entry(PROC\_ENTRY\_FILENAME, &proc\_root);

printk(KERN\_ALERT "Error: Could not initialize /proc/test\n");

return −ENOMEM;

}

Our\_Proc\_File−>owner = THIS\_MODULE;

Our\_Proc\_File−>proc\_iops = &Inode\_Ops\_4\_Our\_Proc\_File;

Our\_Proc\_File−>proc\_fops = &File\_Ops\_4\_Our\_Proc\_File;

Our\_Proc\_File−>mode = S\_IFREG | S\_IRUGO | S\_IWUSR;

Our\_Proc\_File−>uid = 0;

Our\_Proc\_File−>gid = 0;

Our\_Proc\_File−>size = 80;

printk(KERN\_INFO "/proc/test created\n");

return 0;

}

/\*

\* Cleanup − unregister our file from /proc. This could get dangerous if

\* there are still processes waiting in WaitQ, because they are inside our

\* open function, which will get unloaded. I'll explain how to avoid removal

\* of a kernel module in such a case in chapter 10.

\*/

void cleanup\_module()

{

remove\_proc\_entry(PROC\_ENTRY\_FILENAME, &proc\_root);

printk(KERN\_INFO "/proc/test removed\n");

}

Example 9−2. cat\_noblock.c

/\* cat\_noblock.c − open a file and display its contents, but exit rather than

\* wait for input \*/

/\* Copyright (C) 1998 by Ori Pomerantz \*/

#include <stdio.h> /\* standard I/O \*/

#include <fcntl.h> /\* for open \*/

#include <unistd.h> /\* for read \*/

#include <stdlib.h> /\* for exit \*/

#include <errno.h> /\* for errno \*/

#define MAX\_BYTES 1024\*4

main(int argc, char \*argv[])

{

int fd; /\* The file descriptor for the file to read \*/

size\_t bytes; /\* The number of bytes read \*/

char buffer[MAX\_BYTES]; /\* The buffer for the bytes \*/

/\* Usage \*/

if (argc != 2) {

printf("Usage: %s <filename>\n", argv[0]);

puts("Reads the content of a file, but doesn't wait for input");

exit(−1);

}

/\* Open the file for reading in non blocking mode \*/

fd = open(argv[1], O\_RDONLY | O\_NONBLOCK);

/\* If open failed \*/

if (fd == −1) {

if (errno = EAGAIN)

puts("Open would block");

else

puts("Open failed");

exit(−1);

}

/\* Read the file and output its contents \*/

do {

int i;

/\* Read characters from the file \*/

bytes = read(fd, buffer, MAX\_BYTES);

/\* If there's an error, report it and die \*/

if (bytes == −1) {

if (errno = EAGAIN)

puts("Normally I'd block, but you told me not to");

else

puts("Another read error");

exit(−1);

}

/\* Print the characters \*/

if (bytes > 0) {

for(i=0; i<bytes; i++)

putchar(buffer[i]);

}

/\* While there are no errors and the file isn't over \*/

} while (bytes > 0);

}