**Chapter 5. The /proc File System**

**5.1. The /proc File System**

In Linux, there is an additional mechanism for the kernel and kernel modules to send information to processes − the /proc file system. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as /proc/modules which provides the list of modules and /proc/meminfo which stats memory usage statistics.

The method to use the proc file system is very similar to the one used with device drivers − a structure is created with all the information needed for the /proc file, including pointers to any handler functions (in our case there is only one, the one called when somebody attempts to read from the /proc file). Then, init\_module registers the structure with the kernel and cleanup\_module unregisters it.

The reason we use proc\_register\_dynamic[8] is because we don't want to determine the inode number used for our file in advance, but to allow the kernel to determine it to prevent clashes. Normal file systems are located on a disk, rather than just in memory (which is where /proc is), and in that case the inode number is a pointer to a disk location where the file's index−node (inode for short) is located. The inode contains information about the file, for example the file's permissions, together with a pointer to the disk location or locations where the file's data can be found.

Because we don't get called when the file is opened or closed, there's nowhere for us to put try\_module\_get and try\_module\_put in this module, and if the file is opened and then the module is removed, there's no way to avoid the consequences.

Here a simple example showing how to use a /proc file. This is the HelloWorld for the /proc filesystem. There are three parts: create the file /proc/helloworld in the function init\_module, return a value (and a buffer) when the file /proc/helloworld is read in the callback function procfs\_read, and delete the file /proc/helloworld in the function cleanup\_module.

The /proc/helloworld is created when the module is loaded with the function create\_proc\_entry. The return value is a 'struct proc\_dir\_entry \*', and it will be used to configure the file /proc/helloworld (for example, the owner of this file). A null return value means that the creation has failed.

Each time, everytime the file /proc/helloworld is read, the function procfs\_read is called. Two parameters of this function are very important: the buffer (the first parameter) and the offset (the third one). The content of the buffer will be returned to the application which read it (for example the cat command). The offset is the current position in the file. If the return value of the function isn't null, then this function is called again. So be careful with this function, if it never returns zero, the read function is called endlessly.

% cat /proc/helloworld

HelloWorld!

**Example 5−1. procfs1.c**

/\*

\* procfs1.c − create a "file" in /proc

\*

\*/

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

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

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

#define procfs\_name "helloworld"

/\*\*

\* This structure hold information about the /proc file

\*/

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

/\* Put data into the proc fs file.

\* Arguments

\* =========

\* 1. The buffer where the data is to be inserted, if you decide to use it.

\* 2. A pointer to a pointer to characters. This is useful if you don't want to \* use the buffer allocated by the kernel.

\* 3. The current position in the file

\* 4. The size of the buffer in the first argument.

\* 5. Write a "1" here to indicate EOF.

\* 6. A pointer to data (useful in case one common read for multiple /proc/...

\* entries)

\* Usage and Return Value

\* ======================

\* A return value of zero means you have no further information at this time (end \* of file). A negative return value is an error condition.

\* For More Information

\* ====================

\* The way I discovered what to do with this function wasn't by reading

\* documentation, but by reading the code which used it. I just looked to

\* see what uses the get\_info field of proc\_dir\_entry struct (I used a combination

\* of find and grep, if you're interested), and I saw that it is used in <kernel

\* source directory>/fs/proc/array.c.

\* If something is unknown about the kernel, this is usually the way to go. In

\* Linux we have the great advantage of having the kernel source code for free −

\*use it.

\*/

int procfile\_read(char \*buffer, char \*\*buffer\_location,off\_t offset, int buffer\_length, int \*eof, void \*data)

{

int ret;

printk(KERN\_INFO "procfile\_read (/proc/%s) called\n", procfs\_name);

/\*

\* We give all of our information in one go, so if the user asks us if we

\* have more information the answer should always be no.

\*

\* This is important because the standard read function from the library

\* would continue to issue the read system call until the kernel replies

\* that it has no more information, or until its buffer is filled.

\*/

if (offset > 0) {

/\* we have finished to read, return 0 \*/

ret = 0;

} else {

/\* fill the buffer, return the buffer size \*/

ret = sprintf(buffer, "HelloWorld!\n");

}

return ret;

}

int init\_module()

{

Our\_Proc\_File = create\_proc\_entry(procfs\_name, 0644, NULL);

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

remove\_proc\_entry(procfs\_name, &proc\_root);

printk(KERN\_ALERT "Error: Could not initialize /proc/%s\n",

procfs\_name);

return −ENOMEM;

}

Our\_Proc\_File−>read\_proc = procfile\_read;

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

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

Our\_Proc\_File−>uid = 0;

Our\_Proc\_File−>gid = 0;

Our\_Proc\_File−>size = 37;

printk(KERN\_INFO "/proc/%s created\n", procfs\_name);

return 0; /\* everything is ok \*/

}

void cleanup\_module()

{

remove\_proc\_entry(procfs\_name, &proc\_root);

printk(KERN\_INFO "/proc/%s removed\n", procfs\_name);

}

**5.2. Read and Write a /proc File**

We have seen a very simple example for a /proc file where we only read the file /proc/helloworld. It's also possible to write in a /proc file. It works the same way as read, a function is called when the /proc file is written. But there is a little difference with read, data comes from user, so you have to import data from user space to kernel space (with copy\_from\_user or get\_user).

The reason for copy\_from\_user or get\_user is that Linux memory (on Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes.

The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The put\_user and get\_user macros allow you to access that memory. These functions handle only one caracter, you can handle several caracters with copy\_to\_user and copy\_from\_user. As the buffer (in read or write function) is in kernel space, for write function you need to import data because it comes from user space, but not for the read function because data is already in kernel space.

**Example 5−2. procfs2.c**

/\*\*

\* procfs2.c − create a "file" in /proc

\*/

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

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

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

#include <asm/uaccess.h> /\* for copy\_from\_user \*/

#define PROCFS\_MAX\_SIZE 1024

#define PROCFS\_NAME "buffer1k"

/\*\*

\* This structure hold information about the /proc file

\*/

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

/\*\*

\* The buffer used to store character for this module

\*/

static char procfs\_buffer[PROCFS\_MAX\_SIZE];

/\*\*

\* The size of the buffe

\*/

static unsigned long procfs\_buffer\_size = 0;

/\*\*

\* This function is called then the /proc file is read

\*/

int procfile\_read(char \*buffer, char \*\*buffer\_location, off\_t offset, int buffer\_length, int \*eof, void \*data)

{

int ret;

printk(KERN\_INFO "procfile\_read (/proc/%s) called\n", PROCFS\_NAME);

if (offset > 0) {

/\* we have finished to read, return 0 \*/

ret = 0;

} else {

/\* fill the buffer, return the buffer size \*/

memcpy(buffer, procfs\_buffer, procfs\_buffer\_size);

ret = procfs\_buffer\_size;

}

return ret;

}

/\*\*

\* This function is called with the /proc file is written

\*/

int procfile\_write(struct file \*file, const char \*buffer, unsigned long count,

void \*data)

{

/\* get buffer size \*/

procfs\_buffer\_size = count;

if (procfs\_buffer\_size > PROCFS\_MAX\_SIZE ) {

procfs\_buffer\_size = PROCFS\_MAX\_SIZE;

}

/\* write data to the buffer \*/

if ( copy\_from\_user(procfs\_buffer, buffer, procfs\_buffer\_size) ) {

return −EFAULT;

}

return procfs\_buffer\_size;

}

/\*\*

\*This function is called when the module is loaded

\*/

int init\_module()

{

/\* create the /proc file \*/

Our\_Proc\_File = create\_proc\_entry(PROCFS\_NAME, 0644, NULL);

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

remove\_proc\_entry(PROCFS\_NAME, &proc\_root);

printk(KERN\_ALERT "Error: Could not initialize /proc/%s\n",

PROCFS\_NAME);

return −ENOMEM;

}

Our\_Proc\_File−>read\_proc = procfile\_read;

Our\_Proc\_File−>write\_proc = procfile\_write;

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

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

Our\_Proc\_File−>uid = 0;

Our\_Proc\_File−>gid = 0;

Our\_Proc\_File−>size = 37;

printk(KERN\_INFO "/proc/%s created\n", PROCFS\_NAME);

return 0; /\* everything is ok \*/

}

/\*\*

\*This function is called when the module is unloaded

\*/

void cleanup\_module()

{

remove\_proc\_entry(PROCFS\_NAME, &proc\_root);

printk(KERN\_INFO "/proc/%s removed\n", PROCFS\_NAME);

}

**5.3. Manage /proc file with standard filesystem**

We have seen how to read and write a /proc file with the /proc interface. But it's also possible to manage /proc file with inodes. The main interest is to use advanced function, like permissions.

In Linux, there is a standard mechanism for file system registration. Since every file system has to have its own functions to handle inode and file operations[9], there is a special structure to hold pointers to all those functions, struct inode\_operations, which includes a pointer to struct file\_operations. In /proc, whenever we register a new file, we're allowed to specify which struct inode\_operations will be used to access to it. This is the mechanism we use, a struct inode\_operations which includes a

pointer to a struct file\_operations which includes pointers to our procfs\_read and procfs\_write functions.

Another interesting point here is the module\_permission function. This function is called whenever a process tries to do something with the /proc file, and it can decide whether to allow access or not. Right now it is only based on the operation and the uid of the current user (as available in current, a pointer to a structure which includes information on the currently running process), but it could be based on anything we like, such as what other processes are doing with the same file, the time of day, or the last input we received.

It's important to note that the standard roles of read and write are reversed in the kernel. Read functions are used for output, whereas write functions are used for input. The reason for that is that read and write refer to the user's point of view −−− if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

**Example 5−3. procfs3.c**

/\*

\* procfs3.c − create a "file" in /proc, use the file\_operation way to manage the

\* file.

\*/

#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 <asm/uaccess.h> /\* for copy\_\*\_user \*/

#define PROC\_ENTRY\_FILENAME "buffer2k"

#define PROCFS\_MAX\_SIZE 2048

/\*\*

\* The buffer (2k) for this module

\*/

static char procfs\_buffer[PROCFS\_MAX\_SIZE];

/\*\*

\* The size of the data hold in the buffer

\*/

static unsigned long procfs\_buffer\_size = 0;

/\*\*

\* The structure keeping information about the /proc file

\*/

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

/\*\*

\* This funtion is called when the /proc file is read

\*/

static ssize\_t procfs\_read(struct file \*filp, /\* see include/linux/fs.h \*/

char \*buffer, /\* buffer to fill with data \*/ size\_t length, /\* length of the buffer \*/

loff\_t \* offset)

{

static int finished = 0;

/\*

\* We return 0 to indicate end of file, that we have

\* no more information. Otherwise, processes will

\* continue to read from us in an endless loop.

\*/

if ( finished ) {

printk(KERN\_INFO "procfs\_read: END\n");

finished = 0;

return 0;

}

finished = 1;

/\*

\* We use put\_to\_user to copy the string from the kernel's memory segment to \* the memory segment of the process that called us. get\_from\_user, BTW, is

\* used for the reverse.

\*/

if ( copy\_to\_user(buffer, procfs\_buffer, procfs\_buffer\_size) ) {

return −EFAULT;

}

printk(KERN\_INFO "procfs\_read: read %lu bytes\n", procfs\_buffer\_size);

return procfs\_buffer\_size; /\* Return the number of bytes "read" \*/

}

/\*

\* This function is called when /proc is written

\*/

static ssize\_t procfs\_write(struct file \*file, const char \*buffer, size\_t len, loff\_t \* off)

{

if ( len > PROCFS\_MAX\_SIZE ) {

procfs\_buffer\_size = PROCFS\_MAX\_SIZE;

}

else {

procfs\_buffer\_size = len;

}

if ( copy\_from\_user(procfs\_buffer, buffer, procfs\_buffer\_size) ) {

return −EFAULT;

}

printk(KERN\_INFO "procfs\_write: write %lu bytes\n", procfs\_buffer\_size);

return procfs\_buffer\_size;

}

/\*

\* 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 referece only, and can be overridden here.

\*/

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

{

/\*

\* 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;

}

/\*

\* The file is opened − we don't really care about that, but it does mean we need

\* to increment the module's reference count.

\*/

int procfs\_open(struct inode \*inode, struct file \*file)

{

try\_module\_get(THIS\_MODULE);

return 0;

}

/\*

\* The file is closed − again, interesting only because of the reference count.

\*/

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

{

module\_put(THIS\_MODULE);

return 0; /\* success \*/

}

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

.read = procfs\_read,

.write = procfs\_write,

.open = procfs\_open,

.release = procfs\_close,

};

/\*

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

\*/

int init\_module()

{

/\* create the /proc file \*/

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

/\* check if the /proc file was created successfuly \*/

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

printk(KERN\_ALERT "Error: Could not initialize /proc/%s\n",

PROC\_ENTRY\_FILENAME);

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/%s created\n", PROC\_ENTRY\_FILENAME);

return 0; /\* success \*/

}

void cleanup\_module()

{

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

printk(KERN\_INFO "/proc/%s removed\n", PROC\_ENTRY\_FILENAME);

}

Still hungry for procfs examples? Well, first of all keep in mind, there are rumors around, claiming that procfs is on it's way out, consider using sysfs instead. Second, if you really can't get enough, there's a highly recommendable bonus level for procfs below linux/Documentation/DocBook/ . Use make help in your toplevel kernel directory for instructions about how to convert it into your favourite format. Example: make htmldocs . Consider using this mechanism, in case you want to document something kernel related yourself.

**5.4. Manage /proc file with seq\_file**

As we have seen, writing a /proc file may be quite "complex". So to help people writting /proc file, there is an API named seq\_file that helps formating a /proc file for output. It's based on sequence, which is composed of 3 functions: start(), next(), and stop(). The seq\_file API starts a sequence when a user read the /proc file.

A sequence begins with the call of the function start(). If the return is a non NULL value, the function next() is called. This function is an iterator, the goal is to go thought all the data. Each time next() is called, the function show() is also called. It writes data values in the buffer read by the user. The function next() is called until it returns NULL. The sequence ends when next() returns NULL, then the function stop() is called.

BE CARREFUL: when a sequence is finished, another one starts. That means that at the end of function stop(), the function start() is called again. This loop finishes when the function start() returns NULL. You can see a scheme of this in the figure "How seq\_file works".

Figure 5−1. How seq\_file works

Seq\_file provides basic functions for file\_operations, as seq\_read, seq\_lseek, and some others. But nothing to write in the /proc file. Of course, you can still use the same way as in the previous example.

**Example 5−4. procfs4.c**

/\*\*

\* procfs4.c − create a "file" in /proc

\* This program uses the seq\_file library to manage the /proc file.

\*/

#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/seq\_file.h> /\* for seq\_file \*/

#define PROC\_NAME "iter"

MODULE\_AUTHOR("Philippe Reynes");

MODULE\_LICENSE("GPL");

/\*\*

\* This function is called at the beginning of a sequence.

\* ie, when:

\* − the /proc file is read (first time)

\* − after the function stop (end of sequence)

\*/

static void \*my\_seq\_start(struct seq\_file \*s, loff\_t \*pos)

{

static unsigned long counter = 0;

/\* beginning a new sequence ? \*/

if ( \*pos == 0 )

{

/\* yes => return a non null value to begin the sequence \*/

return &counter;

}

else

{

/\* no => it's the end of the sequence, return end to stop reading \*/

\*pos = 0;

return NULL;

}

}

/\*\*

\* This function is called after the beginning of a sequence.

\* It's called untill the return is NULL (this ends the sequence).

\*/

static void \*my\_seq\_next(struct seq\_file \*s, void \*v, loff\_t \*pos)

{

unsigned long \*tmp\_v = (unsigned long \*)v;

(\*tmp\_v)++;

(\*pos)++;

return NULL;

}

/\*\*

\* This function is called at the end of a sequence

\*/

static void my\_seq\_stop(struct seq\_file \*s, void \*v)

{

/\* nothing to do, we use a static value in start() \*/

}

/\*\*

\* This function is called for each "step" of a sequence

\*/

static int my\_seq\_show(struct seq\_file \*s, void \*v)

{

loff\_t \*spos = (loff\_t \*) v;

seq\_printf(s, "%Ld\n", \*spos);

return 0;

}

/\*\*

\* This structure gather "function" to manage the sequence

\*/

static struct seq\_operations my\_seq\_ops = {

.start = my\_seq\_start,

.next = my\_seq\_next,

.stop = my\_seq\_stop,

.show = my\_seq\_show

};

/\*\*

\* This function is called when the /proc file is open.

\*/

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

{

return seq\_open(file, &my\_seq\_ops);

};

/\*\*

\* This structure gather "function" that manage the /proc file

\*/

static struct file\_operations my\_file\_ops = {

.owner = THIS\_MODULE,

.open = my\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = seq\_release

};

/\*\*

\* This function is called when the module is loaded

\*/

int init\_module(void)

{

struct proc\_dir\_entry \*entry;

entry = create\_proc\_entry(PROC\_NAME, 0, NULL);

if (entry) {

entry−>proc\_fops = &my\_file\_ops;

}

return 0;

}

/\*\*

\* This function is called when the module is unloaded.

\*/

void cleanup\_module(void)

{

remove\_proc\_entry(PROC\_NAME, NULL);

}

If you want more information, you can read this web page:

http://lwn.net/Articles/22355/

http://www.kernelnewbies.org/documents/seq\_file\_howto.txt

You can also read the code of fs/seq\_file.c in the linux kernel.