

Access the Linux kernel using the /proc filesystem

This virtual filesystem opens a window of communication between the kernel and user space

M. Tim Jones (mtj@mtjones.com)

14 March 2006

Consultant Engineer
Emulex

The /proc filesystem is a virtual filesystem that permits a novel approach for communication between the Linux® kernel and user space. In the /proc filesystem, virtual files can be read from or written to as a means of communicating with entities in the kernel, but unlike regular files, the content of these virtual files is dynamically created. This article introduces you to the /proc virtual filesystem and demonstrates its use.

The /proc filesystem was originally developed to provide information on the processes in a system. But given the filesystem's usefulness, many elements of the kernel use it both to report information and to enable dynamic runtime configuration.

The /proc filesystem contains directories (as a way of organizing information) and virtual files. A virtual file can present information from the kernel to the user and also serve as a means of sending information from the user to the kernel. It's not actually required to do both, but this article shows you how to configure the filesystem for input and output.

A short article like this can't detail all the uses of /proc, but it does demonstrate a couple of uses to give you an idea of how powerful /proc can be. Listing 1 is an interactive tour of some of the /proc elements. It shows the root level of the /proc filesystem. Note the series of numbered files on the left. Each of these is a directory representing a process in the system. Because the first process created in GNU/Linux is the `init` process, it has a `process-id` of `1`. Next, performing an `ls` on the directory shows a list of files. Each file provides details on the particular process. For example, to see the command-line entry for `init`, simply `cat` the `cmdline` file.

Some of the other interesting files in /proc are `cpuinfo`, which identifies the type of processor and its speed; `pci`, which shows the devices found on the PCI buses; and `modules`, which identifies the modules that are currently loaded into the kernel.

Listing 1. Interactive tour of /proc

```
[root@plato]# ls /proc
```

```

1      2040  2347  2874  474      fb      mdstat    sys
104    2061  2356  2930  9      filesystems meminfo   sysrq-trigger
113    2073  2375  2933  acpi    fs      misc     sysvipc
1375   21    2409  2934  buddyinfo ide     modules  tty
1395   2189  2445  2935  bus     interrupts mounts    uptime
1706   2201  2514  2938  cmdline iomem   mtrr     version
179    2211  2515  2947  cpuinfo ioports net      vmstat
180    2223  2607  3     crypto  irq     partitions
181    2278  2608  3004  devices kallsyms pci
182    2291  2609  3008  diskstats kcore   self
2      2301  263   3056  dma     kmsg    slabinfo
2015   2311  2805  394   driver  loadavg stat
2019   2337  2821  4     execdomains locks    swaps

[root@plato 1]# ls /proc/1
auxv      cwd      exe  loginuid mem      oom_adj   root  statm  task
cmdline  environ fd     maps    mounts  oom_score stat  status wchan

[root@plato]# cat /proc/1/cmdline

init [5]

[root@plato]#
```

Listing 2 illustrates reading from and then writing to a virtual file in `/proc`. This example checks and then enables IP forwarding within the kernel's TCP/IP stack.

Listing 2. Reading from and writing to `/proc` (configuring the kernel)

```

[root@plato]# cat /proc/sys/net/ipv4/ip_forward
0

[root@plato]# echo "1" > /proc/sys/net/ipv4/ip_forward

[root@plato]# cat /proc/sys/net/ipv4/ip_forward
1

[root@plato]#
```

Alternatively, you could use `sysctl` to configure these kernel items. See the [Resources](#) section for more information on that.

By the way, the `/proc` filesystem isn't the only virtual filesystem in GNU/Linux. One such system, `sysfs`, is similar to `/proc` but a bit more organized (having learned lessons from `/proc`). However, `/proc` is entrenched and therefore, even though `sysfs` has some advantages over it, `/proc` is here

to stay. There's also the *debugfs* filesystem, but it tends to be (as the name implies) more of a debugging interface. An advantage to debugfs is that it's extremely simple to export a single value to user space (in fact, it's a single call).

Introducing kernel modules

Loadable Kernel Modules (LKM) are an easy way to demonstrate the `/proc` filesystem, because they're a novel way to dynamically add or remove code from the Linux kernel. LKMs are also a popular mechanism for device drivers and filesystems in the Linux kernel.

If you've ever recompiled the Linux kernel, you probably found that in the kernel configuration process, many device drivers and other kernel elements are compiled as modules. If a driver is compiled directly into the kernel, its code and static data occupy space even if they're not used. But if the driver is compiled as a module, it requires memory only if memory is needed and subsequently loaded, into the kernel. Interestingly, you won't notice a performance hit for LKMs, so they're a powerful means of creating a lean kernel that adapts to its environment based upon the available hardware and attached devices.

Here's a simple LKM to help you understand how it differs from standard (non-dynamically loadable) code that you'll find in the Linux kernel. Listing 3 presents the simplest LKM. (You can download the sample code for this article from the [Downloads](#) section, below.)

Listing 3 includes the necessary module header (which defines the module APIs, types, and macros). It then defines the license for the module using `MODULE_LICENSE`. Here, it specifies *GPL* to avoid tainting the kernel.

Listing 3 then defines the module `init` and `cleanup` functions. The `my_module_init` function is called when the module is loaded and the function can be used for initialization purposes. The `my_module_cleanup` function is called when the module is being unloaded and is used to free memory and generally remove traces of the module. Note the use of `printk` here: this is the kernel `printf` function. The `KERN_INFO` symbol is a string that you can use to filter information from entering the kernel ring buffer (much like `syslog`).

Finally, Listing 3 declares the entry and exit functions using the `module_init` and `module_exit` macros. This allows you to name the module `init` and `cleanup` functions the way you want but then tell the kernel which functions are the maintenance functions.

Listing 3. A simple but functional LKM (simple-lkm.c)

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

/* Defines the license for this LKM */
MODULE_LICENSE("GPL");

/* Init function called on module entry */
```

```
int my_module_init( void )
{
    printk(KERN_INFO "my_module_init called.  Module is now loaded.\n");

    return 0;
}

/* Cleanup function called on module exit */
void my_module_cleanup( void )
{
    printk(KERN_INFO "my_module_cleanup called.  Module is now unloaded.\n");

    return;
}

/* Declare entry and exit functions */
module_init( my_module_init );
module_exit( my_module_cleanup );
```

Listing 3 is a real LKM, albeit a simple one. Now, let's build it and test it out on a 2.6 kernel. The 2.6 kernel introduces a new method for kernel module building that I find simpler than the older methods. With the file `simple-lkm.c`, create a makefile whose sole content is:

```
obj-m += simple-lkm.o
```

To build the LKM, use the `make` command as shown in Listing 4.

Listing 4. Building an LKM

```
[root@plato]# make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.11'
  CC [M]  /root/projects/misc/module2.6/simple/simple-lkm.o
  Building modules, stage 2.
  MODPOST
  CC      /root/projects/misc/module2.6/simple/simple-lkm.mod.o
  LD [M]  /root/projects/misc/module2.6/simple/simple-lkm.ko
make: Leaving directory `/usr/src/linux-2.6.11'
[root@plato]#
```

The result is `simple-lkm.ko`. The new naming convention helps to distinguish kernel objects (LKMs) from standard objects. You can now load and unload the module and then view its output. To load the module, use the `insmod` command; conversely, to unload the module, use the `rmmod` command. `lsmod` shows the currently loaded LKMs (see Listing 5).

Listing 5. Inserting, checking, and removing an LKM

```
[root@plato]# insmod simple-lkm.ko
[root@plato]# lsmod
Module                Size  Used by
simple_lkm             1536  0
autofs4              26244  0
video                13956  0
button               5264  0
battery              7684  0
ac                   3716  0
yenta_socket         18952  3
rsrc_nonstatic        9472  1 yenta_socket
uhci_hcd             32144  0
i2c_piix4             7824  0
dm_mod               56468  3
[root@plato]# rmmod simple-lkm
[root@plato]#
```

Note that kernel output goes to the kernel ring buffer and not to `stdout`, because `stdout` is process specific. To inspect messages on the kernel ring buffer, you can use the `dmesg` utility (or work through `/proc` itself with the command `cat /proc/kmsg`). Listing 6 shows the output of the last few messages from `dmesg`.

Listing 6. Reviewing the kernel output from the LKM

```
[root@plato]# dmesg | tail -5
cs: IO port probe 0xa00-0xaff: clean.
eth0: Link is down
eth0: Link is up, running at 100Mbit half-duplex
my_module_init called.  Module is now loaded.
my_module_cleanup called.  Module is now unloaded.
[root@plato]#
```

You can see the module's messages in the kernel output. Now let's move beyond this simple example and look at some of the kernel APIs that allow you to develop useful LKMs.

Integrating into the /proc filesystem

The standard APIs that are available to kernel programmers are also available to LKM programmers. It's even possible for an LKM to export new variables and functions that the kernel can use. A complete treatment of the APIs is beyond the scope of this article, so I simply present some of the elements that I use later to demonstrate a more useful LKM.

Creating and removing a /proc entry

To create a virtual file in the /proc filesystem, use the `create_proc_entry` function. This function accepts a file name, a set of permissions, and a location in the /proc filesystem in which the file is to reside. The return value of `create_proc_entry` is a `proc_dir_entry` pointer (or `NULL`, indicating an error in `create`). You can then use the return pointer to configure other aspects of the virtual file, such as the function to call when a read is performed on the file. The prototype for `create_proc_entry` and a portion of the `proc_dir_entry` structure are shown in Listing 7.

Listing 7. Elements for managing a /proc filesystem entry

```
struct proc_dir_entry *create_proc_entry( const char *name, mode_t mode,
                                         struct proc_dir_entry *parent );

struct proc_dir_entry {
    const char *name;    // virtual file name
    mode_t mode;        // mode permissions
    uid_t uid;          // File's user id
    gid_t gid;          // File's group id
    struct inode_operations *proc_iops; // Inode operations functions
    struct file_operations *proc_fops; // File operations functions
    struct proc_dir_entry *parent;    // Parent directory
    ...
    read_proc_t *read_proc;    // /proc read function
    write_proc_t *write_proc;  // /proc write function
    void *data;                // Pointer to private data
    atomic_t count;            // use count
    ...
};

void remove_proc_entry( const char *name, struct proc_dir_entry *parent );
```

Later you see how to use the `read_proc` and `write_proc` commands to plug in functions for reading and writing the virtual file.

To remove a file from `/proc`, use the `remove_proc_entry` function. To use this function, provide the file name string as well as the location of the file in the `/proc` filesystem (its parent). The function prototype is also shown in Listing 7.

The parent argument can be `NULL` for the `/proc` root or a number of other values, depending upon where you want the file to be placed. Table 1 lists some of the other parent `proc_dir_entry`s that you can use, along with their location in the filesystem.

Table 1. Shortcut `proc_dir_entry` variables

<code>proc_dir_entry</code>	Filesystem location
<code>proc_root_fs</code>	<code>/proc</code>
<code>proc_net</code>	<code>/proc/net</code>
<code>proc_bus</code>	<code>/proc/bus</code>
<code>proc_root_driver</code>	<code>/proc/driver</code>

The Write Callback function

You can write to a `/proc` entry (from the user to the kernel) by using a `write_proc` function. This function has this prototype:

```
int mod_write( struct file *filp, const char __user *buff,
               unsigned long len, void *data );
```

The `filp` argument is essentially an open file structure (we'll ignore this). The `buff` argument is the string data being passed to you. The buffer address is actually a user-space buffer, so you won't be able to read it directly. The `len` argument defines how much data in `buff` is being written. The `data` argument is a pointer to the private data (see [Listing 7](#)). In the module, I declare a function of this type to deal with the incoming data.

Linux provides a set of APIs to move data between user space and kernel space. For the `write_proc` case, I use the `copy_from_user` functions to manipulate the user-space data.

The Read Callback function

You can read data from a `/proc` entry (from the kernel to the user) by using the `read_proc` function. This function has the following prototype:

```
int mod_read( char *page, char **start, off_t off,
              int count, int *eof, void *data );
```

The `page` argument is the location into which you write the data intended for the user, where `count` defines the maximum number of characters that can be written. Use the `start` and `off` arguments when returning more than a page of data (typically 4KB). When all the data have been

written, set the `eof` (end-of-file) argument. As with `write`, data represents private data. The page buffer provided here is in kernel space. Therefore, you can write to it without having to invoke `copy_to_user`.

Other useful functions

You can also create directories within the `/proc` filesystem using `proc_mkdir` as well as symlinks with `proc_symlink`. For simple `/proc` entries that require only a read function, use `create_proc_read_entry`, which creates the `/proc` entry and initializes the `read_proc` function in one call. The prototypes for these functions are shown in Listing 8.

Listing 8. Other useful `/proc` functions

```
/* Create a directory in the proc filesystem */
struct proc_dir_entry *proc_mkdir( const char *name,
                                   struct proc_dir_entry *parent );

/* Create a symlink in the proc filesystem */
struct proc_dir_entry *proc_symlink( const char *name,
                                     struct proc_dir_entry *parent,
                                     const char *dest );

/* Create a proc_dir_entry with a read_proc_t in one call */
struct proc_dir_entry *create_proc_read_entry( const char *name,
                                               mode_t mode,
                                               struct proc_dir_entry *base,
                                               read_proc_t *read_proc,
                                               void *data );

/* Copy buffer to user-space from kernel-space */
unsigned long copy_to_user( void __user *to,
                           const void *from,
                           unsigned long n );

/* Copy buffer to kernel-space from user-space */
unsigned long copy_from_user( void *to,
                             const void __user *from,
                             unsigned long n );
```



```

/* Allocate a 'virtually' contiguous block of memory */
void *vmalloc( unsigned long size );

/* Free a vmalloc'd block of memory */
void vfree( void *addr );

/* Export a symbol to the kernel (make it visible to the kernel) */
EXPORT_SYMBOL( symbol );

/* Export all symbols in a file to the kernel (declare before module.h) */
EXPORT_SYMTAB

```

Fortune cookies through the /proc filesystem

Here's an LKM that supports both reading and writing. This simple application provides a fortune cookie dispenser. After the module is loaded, the user can load text fortunes into it using the `echo` command and then read them back out individually using the `cat` command.

Listing 9 presents the basic module functions and variables. The `init` function (`init_fortune_module`) allocates space for the cookie pot with `vmalloc` and then clears it out with `memset`. With the `cookie_pot` allocated and empty, I create my `proc_dir_entry` next in the `/proc` root called `fortune`. With `proc_entry` successfully created, I initialize my local variables and the `proc_entry` structure. I load my `/proc` `read` and `write` functions (shown in Listings 9 and 10) and identify the owner of the module. The `cleanup` function simply removes the entry from the `/proc` filesystem and then frees the memory that `cookie_pot` occupies.

The `cookie_pot` is a page in length (4KB) and is managed by two indexes. The first, `cookie_index`, identifies where the next cookie will be written. The variable `next_fortune` identifies where the next cookie will be read for output. I simply wrap `next_fortune` to the beginning when all fortunes have been read.

Listing 9. Module init/cleanup and variables

```

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/proc_fs.h>

#include <linux/string.h>

#include <linux/vmalloc.h>

#include <asm/uaccess.h>

```

```
MODULE_LICENSE("GPL");

MODULE_DESCRIPTION("Fortune Cookie Kernel Module");

MODULE_AUTHOR("M. Tim Jones");


#define MAX_COOKIE_LENGTH      PAGE_SIZE

static struct proc_dir_entry *proc_entry;


static char *cookie_pot; // Space for fortune strings
static int cookie_index; // Index to write next fortune
static int next_fortune; // Index to read next fortune


int init_fortune_module( void )
{
    int ret = 0;

    cookie_pot = (char *)vmalloc( MAX_COOKIE_LENGTH );

    if (!cookie_pot) {
        ret = -ENOMEM;
    } else {

        memset( cookie_pot, 0, MAX_COOKIE_LENGTH );

        proc_entry = create_proc_entry( "fortune", 0644, NULL );

        if (proc_entry == NULL) {

            ret = -ENOMEM;
            vfree(cookie_pot);
            printk(KERN_INFO "fortune: Couldn't create proc entry\n");

        } else {
```

```
    cookie_index = 0;
    next_fortune = 0;

    proc_entry->read_proc = fortune_read;
    proc_entry->write_proc = fortune_write;
    proc_entry->owner = THIS_MODULE;
    printk(KERN_INFO "fortune: Module loaded.\n");

}

}

return ret;
}

void cleanup_fortune_module( void )
{
    remove_proc_entry("fortune", &proc_root);
    vfree(cookie_pot);
    printk(KERN_INFO "fortune: Module unloaded.\n");
}

module_init( init_fortune_module );
module_exit( cleanup_fortune_module );
```

Writing a new cookie to the pot is a simple process (shown in Listing 10). With the length of the cookie being written, I check to see that space is available for it. If not, I return `-ENOSPC`, which is communicated to the user process. Otherwise, the space exists, and I use `copy_from_user` to copy the user buffer directly into the `cookie_pot`. I then increment the `cookie_index` (based upon the length of the user buffer) and NULL terminate the string. Finally, I return the number of characters actually written into the `cookie_pot` that is propagated to the user process.

Listing 10. Function to write a fortune

```
ssize_t fortune_write( struct file *filp, const char __user *buff,
```

```
        unsigned long len, void *data )
{
    int space_available = (MAX_COOKIE_LENGTH-cookie_index)+1;

    if (len > space_available) {

        printk(KERN_INFO "fortune: cookie pot is full!\n");
        return -ENOSPC;

    }

    if (copy_from_user( &cookie_pot[cookie_index], buff, len )) {
        return -EFAULT;
    }

    cookie_index += len;
    cookie_pot[cookie_index-1] = 0;

    return len;
}
```

Reading a fortune is just as simple, as shown in Listing 11. Because the buffer that I'll write to (page) is already in kernel space, I can manipulate it directly and use `sprintf` to write the next fortune. If the `next_fortune` index is greater than the `cookie_index` (next position to write), I wrap `next_fortune` back to zero, which is the index of the first fortune. After the fortune is written to the user buffer, I increment the `next_fortune` index by the length of the last fortune written. This places me at the index of the next available fortune. The length of the fortune is returned and propagated to the user.

Listing 11. Function to read a fortune

```
int fortune_read( char *page, char **start, off_t off,
                 int count, int *eof, void *data )
{
    int len;

    if (off > 0) {
```

```
*eof = 1;

return 0;

}

/* Wrap-around */
if (next_fortune >= cookie_index) next_fortune = 0;

len = sprintf(page, "%s\n", &cookie_pot[next_fortune]);

next_fortune += len;

return len;
}
```

You can see from this simple example that communicating with the kernel through the /proc filesystem is a trivial effort. Now take a look at the fortune module in action (Listing 12).

Listing 12. Demonstrating the fortune cookie LKM

```
[root@plato]# insmod fortune.ko
[root@plato]# echo "Success is an individual proposition.  Thomas Watson" > /proc/fortune
[root@plato]# echo "If a man does his best, what else is there?  Gen. Patton" > /proc/fortune
[root@plato]# echo "Cats: All your base are belong to us.  Zero Wing" > /proc/fortune
[root@plato]# cat /proc/fortune
Success is an individual proposition.  Thomas Watson
[root@plato]# cat /proc/fortune
If a man does his best, what else is there?  Gen. Patton
[root@plato]#
```

The /proc virtual filesystem is widely used to report kernel information and also for dynamic configuration. You'll find it integral to both driver and module programming. You can learn more about it in the [Resources](#) below.

Resources

Learn

- "[Administer Linux on the fly](#)" (developerWorks, May 2003) gives you a thorough grounding in /proc, including how you can administer many details of the operating system without ever having to shut down and reboot the machine.
- Explore the [files and subdirectories](#) in the /proc filesystem.
- This article on [driver porting](#) to the 2.6 Linux kernel discusses kernel modules in detail.
- [LinuxHQ](#) is a great site for information on the Linux kernel.
- The [debugfs](#) filesystem is a debugging alternative to /proc.
- "[Kernel comparison: Improvements in kernel development from 2.4 to 2.6](#)" (developerWorks, February 2004) takes a look behind the scenes at the tools, tests, and techniques that make up kernel 2.6.
- "[Kernel debugging with Kprobes](#)" (developerWorks, August 2004) shows how in combination with 2.6 kernels, Kprobes provides a lightweight, non-disruptive, and powerful mechanism to insert the `printk` function dynamically.
- The `printk` function and `dmesg` methods are common means for kernel debugging. Alessandro Rubini's book *Linux Device Drivers* provides an [online chapter](#) about kernel debugging techniques.
- The [sysctl](#) command is another option for dynamic kernel configuration.
- In the [developerWorks Linux zone](#), find more resources for Linux developers.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- [kernel.org](#) has the latest Linux kernel.
- The [GNU make utility documentation](#) is at the [gnu.org site](#).
- The [Modutils](#) package provides a number of utilities for kernel modules.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

M. Tim Jones



M. Tim Jones is an embedded software engineer and the author of *GNU/Linux Application Programming*, *AI Application Programming*, and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a senior principal engineer at Emulex Corp.

© Copyright IBM Corporation 2006

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)