Linux Kernel Procfs Guide

Erik (J.A.K.) Mouw

<mouw@nl.linux.org>

This software and documentation were written while working on the LART computing board (http://www.lartmaker.nl/), which was sponsored by the Delt University of Technology projects Mobile Multi-media Communications and Ubiquitous Communications.

Copyright © 2001 Erik Mouw

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Revision History	
Revision 1.0	May 30, 2001
Initial revision posted to linux-kernel	
Revision 1.1	June 3, 2001
Revised after comments from linux-kernel	

Table of Contents

Preface

This guide describes the use of the procfs file system from within the Linux kernel. The idea to write this guide came up on the #kernelnewbies IRC channel (see http://www.kernelnewbies.org/), when Jeff Garzik explained the use of procfs and forwarded me a message Alexander Viro wrote to the linux-kernel mailing list. I agreed to write it up nicely, so here it is.

I'd like to thank Jeff Garzik <jgarzik@pobox.com> and Alexander Viro <viro@parcelfarce.linux.theplanet.co.uk> for their input, Tim Waugh <twaugh@redhat.com> for his Selfdocbook, and Marc Joosen <marcj@historia.et.tudelft.nl> for proofreading.

Erik

Chapter 1. Introduction

The `/proc` file system (procfs) is a special file system in the linux kernel. It's a virtual file system: it is not associated with a block device but exists only in memory. The files in the procfs are there to allow userland programs access to certain information from the kernel (like process information in `/proc/[0-9]+/`), but also for debug purposes (like `/proc/ksyms`).

This guide describes the use of the procfs file system from within the Linux kernel. It starts by introducing all relevant functions to manage the files within the file system. After that it shows how to communicate with userland, and some tips and tricks will be pointed out. Finally a complete example will be shown.

Note that the files in `/proc/sys` are sysctl files: they don't belong to procfs and are governed by a completely different API described in the Kernel API book.

Chapter 2. Managing procfs entries

Table of Contents

This chapter describes the functions that various kernel components use to populate the procfs with files, symlinks, device nodes, and directories.

A minor note before we start: if you want to use any of the procfs functions, be sure to include the correct header file! This should be one of the first lines in your code:

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

Creating a regular file

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

const char*            name;
mode_t                 mode;
struct proc_dir_entry* parent;
```

This function creates a regular file with the name `name`, file mode `mode` in the directory `parent`. To create a file in the root of the procfs, use `NULL` as `parent` parameter. When successful, the function will return a pointer to the freshly created struct proc_dir_entry; otherwise it will return `NULL`. Chapter 3, *Communicating with userland* describes how to do something useful with regular files.

Note that it is specifically supported that you can pass a path that spans multiple directories. For example `create_proc_entry("drivers/via0/info")` will create the `via0` directory if necessary, with standard `0755` permissions.

If you only want to be able to read the file, the function `create_proc_read_entry` described in the section called “Convenience functions” may be used to create and initialise the procfs entry in one single call.

Creating a symlink

```
struct proc_dir_entry* proc_symlink(name,
                                    parent,
                                    dest);

const char*            name;
struct proc_dir_entry* parent;
const char*            dest;
```

This creates a symlink in the procfs directory `parent` that points from `name` to `dest`. This translates in userland to `ln -s dest name`.

Creating a directory

```
struct proc_dir_entry* proc_mkdir(name,
                                  parent);

const char*            name;
struct proc_dir_entry* parent;
```

Create a directory *name* in the procfs directory *parent*.

Removing an entry

```
void remove_proc_entry(name,
                       parent);

const char*             name;
struct proc_dir_entry*  parent;
```

Removes the entry *name* in the directory *parent* from the procfs. Entries are removed by their *name*, not by the struct proc_dir_entry returned by the various create functions. Note that this function doesn't recursively remove entries.

Be sure to free the *data* entry from the struct proc_dir_entry before `remove_proc_entry` is called (that is: if there was some *data* allocated, of course). See the section called "A single call back for many files" for more information on using the *data* entry.

Chapter 3. Communicating with userland

Table of Contents

Instead of reading (or writing) information directly from kernel memory, procfs works with *call back functions* for files: functions that are called when a specific file is being read or written. Such functions have to be initialised after the procfs file is created by setting the *read_proc* and/or *write_proc* fields in the struct proc_dir_entry* that the function `create_proc_entry` returned:

```
struct proc_dir_entry* entry;

entry->read_proc = read_proc_foo;
entry->write_proc = write_proc_foo;
```

If you only want to use a the *read_proc*, the function `create_proc_read_entry` described in the section called "Convenience functions" may be used to create and initialise the procfs entry in one single call.

Reading data

The read function is a call back function that allows userland processes to read data from the kernel. The read function should have the following format:

```
int read_func(buffer,
              start,
              off,
              count,
```

```
            peof,
            data);
```

```
char*   buffer;
char**  start;
off_t   off;
int     count;
int*    peof;
void*   data;
```

The read function should write its information into the *buffer*, which will be exactly `PAGE_SIZE` bytes long.

The parameter *peof* should be used to signal that the end of the file has been reached by writing `1` to the memory location *peof* points to.

The *data* parameter can be used to create a single call back function for several files, see the section called "A single call back for many files".

The rest of the parameters and the return value are described by a comment in `fs/proc/generic.c` as follows:

> You have three ways to return data:
>
> 1. Leave `*start = NULL`. (This is the default.) Put the data of the requested offset at that offset within the buffer. Return the number (`n`) of bytes there are from the beginning of the buffer up to the last byte of data. If the number of supplied bytes (`= n - offset`) is greater than zero and you didn't signal eof and the reader is prepared to take more data you will be called again with the requested offset advanced by the number of bytes absorbed. This interface is useful for files no larger than the buffer.
>
> 2. Set `*start` to an unsigned long value less than the buffer address but greater than zero. Put the data of the requested offset at the beginning of the buffer. Return the number of bytes of data placed there. If this number is greater than zero and you didn't signal eof and the reader is prepared to take more data you will be called again with the requested offset advanced by `*start`. This interface is useful when you have a large file consisting of a series of blocks which you want to count and return as wholes. (Hack by Paul.Russell@rustcorp.com.au)
>
> 3. Set `*start` to an address within the buffer. Put the data of the requested offset at `*start`. Return the number of bytes of data placed there. If this number is greater than zero and you didn't signal eof and the reader is prepared to take more data you will be called again with the requested offset advanced by the number of bytes absorbed.

Chapter 5, *Example* shows how to use a read call back function.

Writing data

The write call back function allows a userland process to write data to the kernel, so it has some kind of control over the kernel. The write function should have the following format:

int **write_func**(*file*,
buffer,
count,
data);

```
struct file*   file;
const char*    buffer;
unsigned long  count;
void*          data;
```

The write function should read *count* bytes at maximum from the *buffer*. Note that the *buffer* doesn't live in the kernel's memory space, so it should first be copied to kernel space with `copy_from_user`. The *file* parameter is usually ignored. the section called “A single call back for many files” shows how to use the *data* parameter.

Again, Chapter 5, *Example* shows how to use this call back function.

A single call back for many files

When a large number of almost identical files is used, it's quite inconvenient to use a separate call back function for each file. A better approach is to have a single call back function that distinguishes between the files by using the *data* field in struct proc_dir_entry. First of all, the *data* field has to be initialised:

```
struct proc_dir_entry* entry;
struct my_file_data *file_data;

file_data = kmalloc(sizeof(struct my_file_data), GFP_KERNEL);
entry->data = file_data;
```

The *data* field is a void *, so it can be initialised with anything.

Now that the *data* field is set, the `read_proc` and `write_proc` can use it to distinguish between files because they get it passed into their *data* parameter:

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

        if(data == file_data) {
                /* special case for this file */
        } else {
                /* normal processing */
        }

        return len;
}
```

Be sure to free the *data* data field when removing the procfs entry.

Chapter 4. Tips and tricks

Table of Contents

Convenience functions

```
struct proc_dir_entry* create_proc_read_entry(name,
                                              mode,
                                              parent,
                                              read_proc,
                                              data);
```

```
const char*             name;
mode_t                  mode;
struct proc_dir_entry*  parent;
read_proc_t*            read_proc;
void*                   data;
```

This function creates a regular file in exactly the same way as `create_proc_entry` from the section called “Creating a regular file” does, but also allows to set the read function *`read_proc`* in one call. This function can set the *`data`* as well, like explained in the section called “A single call back for many files”.

Modules

If procfs is being used from within a module, be sure to set the *`owner`* field in the struct proc_dir_entry to `THIS_MODULE`.

```
struct proc_dir_entry* entry;

entry->owner = THIS_MODULE;
```

Mode and ownership

Sometimes it is useful to change the mode and/or ownership of a procfs entry. Here is an example that shows how to achieve that:

```
struct proc_dir_entry* entry;

entry->mode =  S_IWUSR |S_IRUSR | S_IRGRP | S_IROTH;
entry->uid = 0;
entry->gid = 100;
```

Chapter 5. Example

```
/*
 * procfs_example.c: an example proc interface
 *
```

```
 * Copyright (C) 2001, Erik Mouw (mouw@nl.linux.org)
 *
 * This file accompanies the procfs-guide in the Linux kernel
 * source. Its main use is to demonstrate the concepts and
 * functions described in the guide.
 *
 * This software has been developed while working on the LART
 * computing board (http://www.lartmaker.nl), which was sponsored
 * by the Delt University of Technology projects Mobile Multi-media
 * Communications and Ubiquitous Communications.
 *
 * This program is free software; you can redistribute
 * it and/or modify it under the terms of the GNU General
 * Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your
 * option) any later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the implied
 * warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
 * PURPOSE.  See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place,
 * Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/jiffies.h>
#include <asm/uaccess.h>


#define MODULE_VERS "1.0"
#define MODULE_NAME "procfs_example"

#define FOOBAR_LEN 8

struct fb_data_t {
        char name[FOOBAR_LEN + 1];
        char value[FOOBAR_LEN + 1];
};


static struct proc_dir_entry *example_dir, *foo_file,
        *bar_file, *jiffies_file, *symlink;


struct fb_data_t foo_data, bar_data;


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

```
        len = sprintf(page, "jiffies = %ld\n",
                      jiffies);

        return len;
}


static int proc_read_foobar(char *page, char **start,
                            off_t off, int count,
                            int *eof, void *data)
{
        int len;
        struct fb_data_t *fb_data = (struct fb_data_t *)data;

        /* DON'T DO THAT - buffer overruns are bad */
        len = sprintf(page, "%s = '%s'\n",
                      fb_data->name, fb_data->value);

        return len;
}


static int proc_write_foobar(struct file *file,
                             const char *buffer,
                             unsigned long count,
                             void *data)
{
        int len;
        struct fb_data_t *fb_data = (struct fb_data_t *)data;

        if(count > FOOBAR_LEN)
                len = FOOBAR_LEN;
        else
                len = count;

        if(copy_from_user(fb_data->value, buffer, len))
                return -EFAULT;

        fb_data->value[len] = '\0';

        return len;
}


static int __init init_procfs_example(void)
{
        int rv = 0;

        /* create directory */
        example_dir = proc_mkdir(MODULE_NAME, NULL);
        if(example_dir == NULL) {
                rv = -ENOMEM;
                goto out;
        }
        /* create jiffies using convenience function */
        jiffies_file = create_proc_read_entry("jiffies",
                                              0444, example_dir,
                                              proc_read_jiffies,
                                              NULL);
        if(jiffies_file == NULL) {
```

```
                rv  = -ENOMEM;
                goto no_jiffies;
        }

        /* create foo and bar files using same callback
         * functions
         */
        foo_file = create_proc_entry("foo", 0644, example_dir);
        if(foo_file == NULL) {
                rv = -ENOMEM;
                goto no_foo;
        }

        strcpy(foo_data.name, "foo");
        strcpy(foo_data.value, "foo");
        foo_file->data = &foo_data;
        foo_file->read_proc = proc_read_foobar;
        foo_file->write_proc = proc_write_foobar;

        bar_file = create_proc_entry("bar", 0644, example_dir);
        if(bar_file == NULL) {
                rv = -ENOMEM;
                goto no_bar;
        }

        strcpy(bar_data.name, "bar");
        strcpy(bar_data.value, "bar");
        bar_file->data = &bar_data;
        bar_file->read_proc = proc_read_foobar;
        bar_file->write_proc = proc_write_foobar;

        /* create symlink */
        symlink = proc_symlink("jiffies_too", example_dir,
                               "jiffies");
        if(symlink == NULL) {
                rv = -ENOMEM;
                goto no_symlink;
        }

        /* everything OK */
        printk(KERN_INFO "%s %s initialised\n",
               MODULE_NAME, MODULE_VERS);
        return 0;

no_symlink:
        remove_proc_entry("bar", example_dir);
no_bar:
        remove_proc_entry("foo", example_dir);
no_foo:
        remove_proc_entry("jiffies", example_dir);
no_jiffies:
        remove_proc_entry(MODULE_NAME, NULL);
out:
        return rv;
}


static void __exit cleanup_procfs_example(void)
{
        remove_proc_entry("jiffies_too", example_dir);
        remove_proc_entry("bar", example_dir);
```

```
        remove_proc_entry("foo", example_dir);
        remove_proc_entry("jiffies", example_dir);
        remove_proc_entry(MODULE_NAME, NULL);

        printk(KERN_INFO "%s %s removed\n",
               MODULE_NAME, MODULE_VERS);
}


module_init(init_procfs_example);
module_exit(cleanup_procfs_example);

MODULE_AUTHOR("Erik Mouw");
MODULE_DESCRIPTION("procfs examples");
MODULE_LICENSE("GPL");
```