Linux Kernel Procfs Guide

Erik (J.A.K.) Mouw
Delft University of Technology
Faculty of Information Technology and Systems

J.A.K.Mouw@its.tudelft.nl
PO BOX 5031
2600 GA
Delft
The Netherlands

Linux Kernel Procfs Guide

by Erik (J.A.K.) Mouw

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@mandrakesoft.com> and Alexander Viro <viro@math.psu.edu> for their input, Tim Waugh <twaugh@redhat.com> for his Selfdocbook (http://people.redhat.com/twaugh/docbook/selfdocbook/), and Marc Joosen <marcj@historia.et.tudelft.nl> for proofreading.

This documentation was written while working on the LART computing board (http://www.lart.tudelft.nl/), which is sponsored by the Mobile Multi-media Communications (http://www.mmc.tudelft.nl/) and Ubiquitous Communications (http://www.ubicom.tudelft.nl/) projects.

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

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
```

2.1. Creating a regular file

```
struct proc_dir_entry* create_proc_entry (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`. 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 may be used to create and initialise the procfs entry in one single call.

2.2. Creating a symlink

```
struct proc_dir_entry* proc_symlink (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*.

2.3. Creating a device

```
struct proc_dir_entry* proc_mknod (const char* name); mode_t
mode); struct proc_dir_entry* parent); kdev_t rdev);
```

Creates a device file *name* with mode *mode* in the procfs directory *parent*. The device file will work on the device *rdev*, which can be generated by using the `MKDEV` macro from `linux/kdev_t.h`. The *mode* parameter *must* contain `S_IFBLK` or `S_IFCHR` to create a device node. Compare with userland `mknod --mode=`*mode name rdev*.

2.4. Creating a directory

```
struct proc_dir_entry* proc_mkdir (const char* name); struct
proc_dir_entry* parent);
```

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

2.5. Removing an entry

```
void remove_proc_entry (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 for more information on using the *`data`* entry.

Chapter 3. Communicating with userland

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;
```