Guía del Procfs del Núcleo Linux

Erik (J.A.K.) Mouw
Universidad de Tecnología Delft
Facultad de Sistemas y Tecnología de la Información

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

Guía del Procfs del Núcleo Linux

por Erik (J.A.K.) Mouw

Historial de revisiones

Revisión 1.0 30 de Mayo, 2001
Revisión Inicial mandada a linux-kernel
Revisión 1.1 3 de Junio, 2001
Revisada después de los comentarios de linux-kernel

Tabla de contenidos

Prefacio

Esta guía describe el uso del sistema de archivos procfs del Núcleo Linux. La idea de escribir esta guía vino del canal de IRC kernelnewbies (ver http://www.kernelnewbies.org/), cuando Jeff Garzik explicó el uso de procfs y me envió un mensaje de Alexander Viro que escribió a la lista de correo linux-kernel. Estuve de acuerdo en escribirla de forma más bonita, por lo tanto aquí está.

Me gustaría dar las gracias a Jeff Garzik <jgarzik@mandrakesoft.com> y Alexander Viro <viro@math.psu.edu> por su entrada, a Tim Waugh <twaugh@redhat.com> por su Selfdocbook (http://people.redhat.com/twaugh/docbook/selfdocbook/), y a Marc Joosen <marcj@historia.et.tudelft.nl> por la profunda lectura.

Esta documentación fue escrita mientras trabajaba en el LART computing board (http://www.lart.tudelft.nl/), que es patrocinada por los proyectos Mobile Multi-media Communications (http://www.mmc.tudelft.nl/) y Ubiquitous Communications (http://www.ubicom.tudelft.nl/).

Erik

Capítulo 1. Introducción

El sistema de archivos `/proc` (procfs) es un sistema de archivos especial en el núcleo Linux. Es un sistema de archivos virtual; no está asociado con un dispositivo de bloque ya que existe sólo en memoria. Los archivos en el procfs están allí para permitir a los programas de usuario acceder a cierta información del núcleo (como información sobre los procesos en `/proc/[0-9]+/`), y también para propósitos de depuración (como `/proc/ksyms`).

Esta guía describe el uso del sistema de archivos procfs del núcleo Linux. Empieza introduciendo todas las funciones relevantes para administar los archivos en el sistema de archivos. Después de esto muestra cómo se comunica con los programas de usuario, y algunos trucos y propinas serán apuntados. Finalmente, será mostrado un ejemplo completo.

Destacar que los archivos en `/proc/sys` son archivos sysctl: no pertenecen al procfs y son gobernados por una API completamente diferente descrita en el libro de la API del Núcleo.

Capítulo 2. Administrando entradas procfs

Este capítulo describe las funciones de varios componentes del núcleo usadas para propagar el procfs con archivos, enlaces simbólicos, nodos de dispositivos, y directorios.

Una pequeña nota antes de empezar: si quieres usar alguna de las funciones procfs, ¡asegúrate de incluir el archivo de cabeceras correspondiente! Esta debería de ser una de las primeras líneas en tu código:

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

2.1. Creando un archivo normal

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

Esta función crea un archivo normal con el nombre *name*, el modo del archivo *mode* en el directorio *parent*. Para crear un archivo en el directorio raíz, usa `NULL` como parámetro a *parent* Cuando tenga éxito, la función retornará un puntero a la nueva struct proc_dir_entry creada; en otro caso retornará `NULL`. Capítulo 3 describe cómo hacer algo útil con los archivos normales.

Destacar que está específicamente soportado el poder pasarle un camino que se extienda a través de múltiples directorios. Por ejemplo, `create_proc_entry(`*`"drivers/via0/info"`*`)` creará el directorio `via0` si es necesario, con los permisos estándar `0755`.

Si sólo quieres ser capaz de leer el archivo, la función `create_proc_read_entry` descrita en Sección 4.1 puede ser usada para crear e inicializar la entrada en el procfs con una simple llamada.

2.2. Creando un enlace simbólico

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

Esto crea un enlace simbólico en el directorio procfs *parent* que apunta a *name* en *dest*. Esto se traduce en los programas de usuario en `ln -s` *dest name*.

2.3. Creando un dispositivo

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

Crea un archivo de dispositivo *`name`* con el modo *`mode`* en el directorio procfs *`parent`*. El archivo del dispositivo trabajará en el dispositivo *`rdev`*, que puede ser generado usando la macro `MKDEV` desde `linux/kdev_t.h`. El parámetro *`mode`* *debe* de contener `S_IFBLK` o `S_IFCHR` para crear un nodo de dispositivo. Compáralo con el programa de usuario `mknod --mode=`*`mode name rdev`*.

2.4. Creando un directorio

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

Crea un directorio *`name`* en el directorio procfs *`parent`*.

2.5. Borrando una entrada

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

Borra la entrada *`name`* en el directorio *`parent`* del procfs. Las entradas son quitadas por su *name*, no por la struct proc_dir_entry retornada por las diversas funciones de creación. Destacar que esta función no borra recursivamente las entradas.

Asegúrate de liberar la entrada *`data`* de struct proc_dir_entry antes de que sea llamado `remove_proc_entry` (esto es: si había alguna *`data`* asignada, por supuesto). Ver Sección 3.3 para más información sobre el uso de la entrada *`data`*.

Capítulo 3. Comunicación con los Procesos de Usuario

En vez de leer (o escribir) información directamente desde la memoria del núcleo, procfs trabaja con *funciones de retrollamada* para los archivos: funciones que son llamadas cuando un archivo específico está siendo leído o escrito. Tales funciones tienen que ser inicializadas después de que el archivo procfs sea creado estableciendo los campos *`read_proc`* y/o *`write_proc`* en la struct proc_dir_entry* que retorna la función `create_proc_entry`:

```
struct proc_dir_entry* entry;

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

Si sólo quieres usar la *`read_proc`*, la función `create_proc_read_entry` descrita en Sección 4.1 puede ser utilizada para crear e inicializar la entrada procfs con una simple llamada.

3.1. Leyendo Datos

La función de lectura es una función de retrollamada que permite a los procesos de usuario leer datos del núcleo. La función de lectura debe de tener el siguiente formato:

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

La función de lectura debería de escribir su información en *`page`*. Para un uso adecuado, la función debería de empezar escribiendo en un desplazamiento de *`off`* en *`page`* y escribir al menos *`count`* bytes, como la mayoría de las funciones de lectura son bastante simples y sólo retornarán una pequeña cantidad de información, estos dos parámetros son usualmente ignorados (y rompe paginadores como `more` y `less`, pero `cat` todavía trabaja).

Si los parámetros *`off`* y *`count`* son usados de una forma adecuada, *`eof`* debería de ser utilizado para señalar que el final del archivo ha llegado escribiendo `1` en la localización de memoria a donde apunta *`eof`*.

El parámetro *`start`* no parece ser usado en ningún sitio en el núcleo. El parámetro *`data`* puede ser usado para crear una función simple de retrollamada para varios archivos, ver Sección 3.3.

La función `read_func` debe de retornar el número de bytes escritos en *`page`*.

Capítulo 5 muestra cómo usar la función de retrollamada de lectura.

3.2. Escribiendo Datos

La función de retrollamada de escritura permite a los procesos de usuario escribir datos en el núcleo, por lo tanto tiene cierta clase de control sobre el núcleo. La función de escritura debería de tener el siguiente formato:

```
int write_func(struct file* file, const char* buffer, unsigned long count, void*
data);
```

La función de escritura lee *`count`* bytes como máximo del *`buffer`*. Destacar que *`buffer`* no reside en el espacio de memoria del núcleo, por lo tanto debería de ser primero copiado al espacio del núcleo con `copy_from_user`. El parámetro *`file`* es usualmente ignorado. Sección 3.3 muestra como usar el parámetro *`data`*.

Otra vez, Capítulo 5 muestra cómo usar esta función de retrollamada.

3.3. Una simple retrollamada para varios archivos

Cuando es usado un gran número de archivos casi idénticos, es bastante conveniente usar una función de retrollamada separada para cada archivo. Una mejor aproximación es tener una función simple de retrollamada que distinga entre los archivos usando el campo *`data`* en struct proc_dir_entry. Lo primero de todo, el campo *`data`* tiene que estar inicializado:

```
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;
```

El campo *`data`* es un void *, por lo tanto puede ser inicializado con cualquier cosa.

Ahora que el campo *`data`* está establecido, `read_proc` y `write_proc` pueden utilizarlo para distinguir entre archivos, porque ellos son pasados en sus parámetros *`data`*:

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

        if(data == file_data) {
                /* caso especial para este archivo */
        } else {
                /* procesamiento normal */
        }

        return len;
}
```

Asegúrate de liberar el campo de datos *`data`* cuando quites la entrada procfs.

Capítulo 4. Trucos y Propinas

4.1. Funciones Convenientes

```
struct proc_dir_entry* create_proc_read_entry(const char* name, mode_t mode, struct
proc_dir_entry* parent, read_proc_t* read_proc, void* data);
```

Esta función crea un archivo normal exactamente de la misma forma que lo hace `create_proc_entry` desde Sección 2.1, pero también permite establecer la función de lectura *`read_proc`* en una llamada. Esta función puede establecer también el parámetro *`data`*, como ya ha sido explicado en Sección 3.3.

4.2. Módulos

SI procfs está siendo usado desde un módulo, asegúrate de establecer el campo *`owner`* en struct proc_dir_entry a `THIS_MODULE`.

```
struct proc_dir_entry* entry;

entry->owner = THIS_MODULE;
```

4.3. Modo y Dueño

Algunas veces es útil cambiar el modo y/o dueño de una entrada procfs. Aquí hay un ejemplo que muestra cómo realizar esto:

```
struct proc_dir_entry* entry;

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

Capítulo 5. Ejemplo

Para compilar este módulo utiliza:

gcc -I/usr/src/linux-2.4/include -Wall -DMODULE -D__KERNEL__ -DLINUX -c ejemplo_procfs.c

```
/*
 * ejemplo_procfs.c: un ejemplo de la interface proc
 *
 * Copyright (C) 2001, Erik Mouw (J.A.K.Mouw@its.tudelft.nl)
 *
 * Este archivo acompaña la guía de procfs en el código del
 * núcleo Linux. Su uso principal es demostrar los conceptos y
 * funciones descritas en la guía.
 *
 * Este software ha sido desarrollado mientras se trabajaba en
 * el LART computing board (http://www.lart.tudelft.nl/), que
 * está patrocinado por los proyectos Mobile Multi-media
 * Communications (http://www.mmc.tudelft.nl/) y Ubiquitous
 * Communications (http://www.ubicom.tudelft.nl/).
 *
 * El autor puede ser encontrado en:
 *
 *  Erik Mouw
 *  Information and Communication Theory Group
 *  Faculty of Information Technology and Systems
 *  Delft University of Technology
 *  P.O. Box 5031
 *  2600 GA Delft
 *  The Netherlands
 *
 *
 * Este programa es software libre; puedes redistribuirlo
 * y/o modificarlo bajo los términos de la GNU General
 * Public License tal como ha sido publicada por la Free
 * Software Foundation; por la versión 2 de la Licencia,
 * o (a tu elección) cualquier versión posterior.
 *
 * Este programa es distribuido con la esperanza de que
 * sea útil, pero SIN NINGUNA GARANTÍA; sin incluso la
 * implicada de COMERCIABILIDAD o ADECUACCION PARA UN
 * PROPOSITO PARTICULAR. Para más detalles refiérase a la
 * GNU General Public License.
 *
 * Deberías de haber recibido una copia de la GNU General
 * Public License con este programa; si no es así, escribe a
 * 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/sched.h>
#include <asm/uaccess.h>


#define MODULE_VERSION "1.0"
#define MODULE_NAME "ejemplo_procfs"

#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, *tty_device, *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;

        MOD_INC_USE_COUNT;

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

        MOD_DEC_USE_COUNT;

        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;

        MOD_INC_USE_COUNT;

        len = sprintf(page, "%s = '%s'\n",
                      fb_data->name, fb_data->value);
```

```
        MOD_DEC_USE_COUNT;

        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;

        MOD_INC_USE_COUNT;

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

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

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

        MOD_DEC_USE_COUNT;

        return len;
}


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

        /* crea directorio */
        example_dir = proc_mkdir(MODULE_NAME, NULL);
        if(example_dir == NULL) {
                rv = -ENOMEM;
                goto out;
        }

        example_dir->owner = THIS_MODULE;

        /* crea jiffies usando la función conveniente */
        jiffies_file = create_proc_read_entry("jiffies",
                                              0444, example_dir,
                                              proc_read_jiffies,
                                              NULL);
        if(jiffies_file == NULL) {
```

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

jiffies_file->owner = THIS_MODULE;

/* crea los archivos foo y bar usando las mismas
 * funciones de retrollamada
 */
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;
foo_file->owner = THIS_MODULE;

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;
bar_file->owner = THIS_MODULE;

/* crea dispositivo tty */
tty_device = proc_mknod("tty", S_IFCHR | 0666,
                        example_dir, MKDEV(5, 0));
if(tty_device == NULL) {
        rv = -ENOMEM;
        goto no_tty;
}

tty_device->owner = THIS_MODULE;

/* crea enlace simbólico */
symlink = proc_symlink("tambien_jiffies", example_dir,
                       "jiffies");
if(symlink == NULL) {
        rv = -ENOMEM;
        goto no_symlink;
}
```

```
        symlink->owner = THIS_MODULE;

        /* todo está OK */
        printk(KERN_INFO "%s %s inicializado\n",
               MODULE_NAME, MODULE_VERSION);
        return 0;

no_symlink:
        remove_proc_entry("tty", example_dir);
no_tty:
        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("tambien_jiffies", example_dir);
        remove_proc_entry("tty", 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 borrado\n",
               MODULE_NAME, MODULE_VERSION);
}


module_init(init_procfs_example);
module_exit(cleanup_procfs_example);

MODULE_AUTHOR("Erik Mouw");
MODULE_DESCRIPTION("ejemplos procfs");

EXPORT_NO_SYMBOLS;
```

Capítulo 6. Sobre la Traducción

Este documento es la traducción de "Linux Kernel Procfs Guide", documento que acompaña al código del núcleo de Linux, versión 2.4.18.

Este documento ha sido traducido por Rubén Melcón <melkon@terra.es>; y es publicado por TLDP-ES (http://es.tldp.org)

Versión de la tradución 0.04 (Julio de 2002).

Si tienes comentarios sobre la traducción, ponte en contacto con Rubén Melcón <melkon@terra.es>