Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2012/2013

Kernel Module

Laden von Kernelmodulen

Linux unterstützt die Auslagerung von Kernel Code in Module. Diese können mit `insmod` dynamisch hinzugeladen werden.

Zum Arbeiten mit Modulen existieren folgende Befehle:

- `insmod` zum Laden eines Moduls in den Kernel,
- `rmmod` zum Entladen eines Moduls,
- `lsmod` zum Auflisten der geladenen Module,
- `modinfo` zum Anzeigen von Infos zu einem Modul.

Hinweis: Beim Laden eines Moduls, das nicht erklärt, unter GPL zu stehen, wird der Kernel als *tainted* markiert (damit gibt es keinen Support seitens der Kernel-Community, weiterhin können Funktionen des Kernels ausschließlich GPL-Modulen zugänglich gemacht werden).

Einfaches Modul

Ein Modul muss mindestens zwei Funktionen aufweisen: je eine, die beim Laden bzw. Entladen ausgeführt wird:

```
/*
 *  hello.c - Demo module
 */
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */
#include <linux/init.h>     /* Needed for the macros */

static int __init init_hello(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

static void __exit cleanup_hello(void)
{
    printk(KERN_INFO "Goodbye, world\n");
}

module_init(init_hello);
module_exit(cleanup_hello);
```

Analyse

Im vorliegenden Beispiel werden die Funktionen `init_hello` und `cleanup_hello` über die Makros `module_init` und `module_exit` als Routinen definiert, die beim Laden bzw. Entladen des Moduls aufgerufen werden.

Hier tun diese beiden Funktionen nichts außer dem Aufruf von `printk`. Dieses Makro (!) ist analog zu `printf`, nur dass die Meldungen dem Kernel-Logger übergeben werden, der sie je nach Schweregrad (hier `KERN_INFO`) nach `/var/log/messages` und/oder auf die Konsole schreibt (oder je nach Konfiguration direkt wegschmeißt).
Achtung: `printk(KERN_INFO "Dies ist ein Text")` ist kein Tippfehler!

Devices

Selbstgeschriebene Kernel-Module sind meist Device-Treiber, die mit einem Device-Knoten unterhalb von `/dev` Ein- und Ausgabeoperationen durchführen.

Ein solcher Device-Knoten wird mit dem Befehl
`mknod` *`name typ major minor`*
wobei der Typ entweder `b` (block device), `c` oder `u` (character device) oder `p` (FIFO) ist. Die Major-Nummer dient zur Identifikation des Kernelmoduls, das zur Ein- bzw. Ausgabe über das Device zuständig ist, die Minor-Nummer wird beim `open` übergeben und kann bei Bedarf vom Kernelmodul verwendet werden.

Beispiel eines Character-Devices

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you've read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>    /* for put_user */

/*
 *  Prototypes - this would normally go in a .h file
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *,
       size_t, loff_t *);
```

Beispiel eines Character-Devices (Forts.)

```
#define SUCCESS 0
#define DEVICE_NAME "chardev"
/* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80      /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major;       /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?
                 * Used to prevent multiple access to device */
static char msg[BUF_LEN];   /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {
    printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
  }

  printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
  printk(KERN_INFO "the driver, create a dev file with\n");
  printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
  printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
  printk(KERN_INFO "the device file.\n");
  printk(KERN_INFO "Remove the device file and module when done.\n");

  return SUCCESS;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
  /*
   * Unregister the device
   */
  unregister_chrdev(Major, DEVICE_NAME);
}

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
  static int counter = 0;

  if (Device_Open)
    return -EBUSY;
```

Beispiel eines Character-Devices (Forts.)

```
  Device_Open++;
  sprintf(msg, "I already told you %d times Hello world!\n", counter++);
  msg_Ptr = msg;
  try_module_get(THIS_MODULE);

  return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
  Device_Open--;    /* We're now ready for our next caller */

  /*
   * Decrement the usage count, or else once you opened the file, you'll
   * never get get rid of the module.
   */
  module_put(THIS_MODULE);

  return 0;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,
        char *buffer, size_t length, loff_t * offset)
{
  /*
   * Number of bytes actually written to the buffer
   */
  int bytes_read = 0;

  /*
   * If we're at the end of the message,
   * return 0 signifying end of file
   */
  if (*msg_Ptr == 0)
    return 0;
```

Beispiel eines Character-Devices (Forts.)

```
  /*
   * Actually put the data into the buffer
   */
  while (length && *msg_Ptr) {

    /*
     * The buffer is in the user data segment, not the kernel
     * segment so "*" assignment won't work.  We have to use
     * put_user which copies data from the kernel data segment to
     * the user data segment.
     */
    put_user(*(msg_Ptr++), buffer++);

    length--;
    bytes_read++;
  }

  /*
   * Most read functions return the number of bytes put into the buffer
   */
  return bytes_read;
}
```

Beispiel eines Character-Devices (Forts.)

```
/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp,
             const char *buff, size_t len, loff_t * off)
{
  printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
  return -EINVAL;
}
```

Nach dem Laden muss natürlich noch das Device angelegt werden mit

`mknod /dev/chardev c xxx 0`

(für die Major-No. s. `/var/log/messages`)

Testen des Character-Devices

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main() {
    int fd = open("/dev/chardev","r");
    if (fd<0) {
        printf("Fehler %d, %s\n", errno, strerror(errno));
        return(1);
    }
    char msg[80];
    int size = read(fd, msg, 79);
    if (size>=0) msg[size] = '\0';
    printf("%d\n%s\n",size,msg);
    size = read(fd, msg, 79);
    if (size>=0) msg[size] = '\0';
    printf("%d\n%s\n",size,msg);
    close(fd);
    return(0);
}
```