Linux Kernel Modul

Um ein Linux Kernel Modul (zum Beispiel einen Gerätetreiber zu entwickeln), werden neben den bisher im Rahmen der Praktika verwendeten Werkzeuge (wie gcc etc.) einige weitere Softwarekomponenten benötigt. Beispielhaft ist im folgenden erklärt, wie unter Debian bzw. Ubuntu Linux eigene Kernel Module entwickelt werden können[1]. Voraussetzung für das Kompilieren sind insbesondere die Kernel Headerdateien, die jeweils für die aktuell auf Ihrem System verwendete Kernel-Version installiert sein müssen. Diese Header (und weitere benötigte Tools) können mit folgendem Befehl installiert werden:

```
# sudo apt-get install build-essential linux-headers-$(uname -r)
```

Der Rumpf eines Kernelmoduls sieht wie nachfolgend aufgeführt aus.

```
/*
 *    lkm.c - Loadable Kernel Module that prints output to the syslog
 *
 *    Background: http://tldp.org/HOWTO/Module-HOWTO/x73.html
 */

// Defining __KERNEL__ and MODULE allows us to access kernel-level code not
// usually available to userspace programs.
#undef __KERNEL__
#define __KERNEL__

#undef MODULE
#define MODULE

// Linux Kernel/LKM headers: module.h is needed by all modules and kernel.h
// is needed for KERN_INFO.
#include <linux/module.h>// included for all kernel modules
#include <linux/kernel.h>// included for KERN_INFO
#include <linux/init.h>// included for __init and __exit macros

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello world!\n");
    return 0; // Non-zero return means that the module couldn't be loaded.
}
```

[1] Hinweise, wie Sie Kernel Module unter anderen Distributionen kompilieren können, finden Sie im Internet. Mit einigen Modifikationen ist der hier dargestellte Ablauf auch für den Mini-PC Raspberry Pi geeignet. Der hier beschriebene Ablauf wurde mit Ubuntu 12.04 getestet

```
static void __exit hello_cleanup(void)
{
    printk(KERN_INFO "Cleaning up module.\n");
}

module_init(hello_init);
module_exit(hello_cleanup);

MODULE_LICENSE("GPL"); /* to avoid tainted flag */
MODULE_AUTHOR("FH BI Minden"); /* Who wrote this module? */
MODULE_DESCRIPTION("Beispieltreiber"); /* What does this module do */
```

Zunächst werden einige `include` Dateien benötigt. Das Modul verfügt lediglich über zwei eigene Funktionen, die `init` und die `exit` Funktion. Diese Funktionen werden beim Laden bzw. Entladen aufgerufen. Damit ist der Funktionsumfang des Moduls vollständig beschrieben: beim Laden des Moduls erfolgt eine Textausgabe an den `syslog` Dämon ("Hello world!"). Beim Entladen erfolgt die Meldung "Cleaning up module.".

Für eine Erklärung der Präprozessormakros wie `MODULE_LICENSE` sei auf Wikipedia und andere Seiten verwiesen.

Das Modul kompilieren

Um das Modul zu kompilieren verwendet man am besten ein Makefile. Ein lauffähiges Beispiel ist hier gegeben. Achten Sie darauf, dass die auf `all:` bzw. `clean:` folgenden Zeilen mit einem Tab beginnen müssen. Das `Makefile` muss sich in demselben Verzeichnis wie die Codedatei `lkm.c` befinden.

```
obj-m := lkm.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Mit `make` wird das Kernelmodul erzeugt (hier: lkm.ko):

```
martin@redstar:~/bs/lkm$ make
make -C /lib/modules/3.2.0-34-generic-pae/build M=/home/martin/bs/lkm modules
make[1]: Entering directory '/usr/src/linux-headers-3.2.0-34-generic-pae'
  CC [M]  /home/martin/bs/lkm/lkm.o
  Building modules, stage 2.
```

```
  MODPOST 1 modules
  CC      /home/martin/bs/lkm/lkm.mod.o
  LD [M]  /home/martin/bs/lkm/lkm.ko
make[1]: Leaving directory '/usr/src/linux-headers-3.2.0-34-generic-pae'
```

Das Modul kann nun mit dem Befehl `sudo insmod lkm.ko` geladen bzw. mit `sudo rmmod lkm` entladen werden. Eine Übersicht aller geladenen Kernel Module liefert der Befehl `lsmod`.

Die Ausgabe der Kernel-Log Messages kann mit dem Befehl `dmesg` erfolgen:

```
<snip>
[  748.993369] Hello world!
[  758.843606] Cleaning up module.
```

Mit den oben genannten Schritten haben Sie erfolgreich ein erstes einfaches Kernel Modul entwickelt.

Im Folgenden wird ein komplexerer Treiber vorgestellt.

Ein-Byte-Speicher als Kernel Modul

Mit diesem Treiber kann ein `char` gelesen bzw. geschrieben werden. Da mit diesem Beispiel keine Spezialhardware angesteuert werden soll, wird ersatzweise der Arbeitsspeicher als Gerät adressiert.

Initialisierung

Kernel Module erfordern ein Grundgerüst an `include` Files. In Ergänzung zum vorangegangenen ersten Treiberbeispiel sind hier weitere Includes nötig:

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of memory.c functions */
int memory_open(struct inode *inode, struct file *filp);
```

```
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);
void memory_exit(void);
int memory_init(void);

/* Structure that declares the usual file */
/* access functions */
struct file_operations memory_fops = {
  read: memory_read,
  write: memory_write,
  open: memory_open,
  release: memory_release
};

/* Declaration of the init and exit functions */
module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */
/* Major number */
int memory_major = 60;
/* Buffer to store data */
char *memory_buffer;
```

Nach den `include` Dateien werden Funktionen deklariert, die später benötigt werden (`open`, `read`, `write`, ...). Das `struct file_operations` definiert Dateioperationen, mit denen auf das Modul zugegriffen werden kann. Dem Kernel werden zudem die `init` und `exit` Funktion bekanntgemacht (hier: `memory_init` und `memory_exit`). Diese Funktionen werden beim Laden bzw. Entaden des Moduls aus dem Speicher vom Kernel aufgerufen. Am Ende des Codeabschnitts werden globale Variablen des Moduls deklariert. Sie finden wie nachfolgend erklärt Verwendung.

Schnittstelle zwischen Userspace und Kernelspace

Der Zugriff auf diesen Treiber erfolgt aus dem User Space über eine spezielle Datei (diese Dateien liegen im `/dev`-Verzeichnis). Um eine eine solche Datei mit einem Kernel Modul zu verbinden, wird eine Kombination aus der `Major` und `Minor` Nummer benutzt. Um eine Datei für den Zugriff auf den Beispieltreiber zu erzeugen, verwenden Sie auf der Kommandozeile den folgenden Befehl `# sudo mknod /dev/memory c 60 0`

Der Buchstabe `c` bedeutet, dass ein Char-Device erzeugt wird. `60` ist die `Major` Nummer, `0` ist die `Minor` Nummer. Die `Major` Nummer dient dazu, das zugehörige Kernel Modul auszuwählen. Die `Minor` Nummer ist ein Übergabeparameter an das Modul (so können mehrere Dateien auf dasselbe Modul zugreifen). Die `Minor` Nummer wird in diesem Beispiel nicht verwendet und wird auf `0` gesetzt.

Im Kernel Modul wird die Verbindung zum Device File hergestellt. Dazu wird die Funktion `register_chrdev` verwendet.

Diese Funktion wird mit drei Argumenten aufgerufen, der `Major` Nummer, einem String mit dem Namen des Moduls und einem `struct file_operations`, das eine Verbindung zu den Dateioperationen herstellt. Der Aufruf der Funktion erfolgt in der `memory_init` Funktion:

```
int memory_init(void) {
  int result;

  /* Registering device */
  result = register_chrdev(memory_major, "memory", &memory_fops);
  if (result < 0) {
    printk(
      "<1>memory: cannot obtain major number %d\n", memory_major);
    return result;
  }

  /* Allocating memory for the buffer */
  memory_buffer = kmalloc(1, GFP_KERNEL);
  if (!memory_buffer) {
    result = -ENOMEM;
    goto fail;
  }
  memset(memory_buffer, 0, 1);

  printk("<1>Inserting memory module\n");
  return 0;

  fail:
    memory_exit();
    return result;
}
```

Hier wird die Funktion `kmalloc` verwendet, um Speicher im Kernel Space zu reservieren. `kmalloc` kann nahezu analog zu `malloc` verwendet werden. Außerdem enthält die Funktion einige Fehlerüberprüfungen und entsprechende Ausgaben für `syslog`.

Das Modul entladen

Um das Modul zu entladen wird die `memory_exit` aufgerufen. Diese ruft dann `unregister_chrdev` auf, um die `Major` Nummer wieder freizugeben.

```
void memory_exit(void) {
  /* Freeing the major number */
```

```
  unregister_chrdev(memory_major, "memory");

  /* Freeing buffer memory */
  if (memory_buffer) {
    kfree(memory_buffer);
  }

  printk("<1>Removing memory module\n");

}
```

Wichtig ist, dass in dieser Funktion auch der vom Modul benötigte Pufferspeicher wieder freigegeben wird.

Öffnen des Moduls als Datei

Analog zum Öffnen von Dateien aus dem User-Space (mit `fopen`) gibt es die Variable `open` in der Struktur `file_operations`. Diese wird beim Funktionsaufruf von `register_chrdev` verwendet. In diesem Fall wird bei einem `open`-Kommando die Funktion `memory_open` aufgerufen. Als Aufrufargumente werden eine inode Struktur, die Informationen über die Major und Minor Nummer enthält sowie eine Struktur mit den nötigen Dateioperationen übergeben. Für eine ausführliche Erklärung wird an dieser Stelle auf Fachliteratur oder eine Recherche in den einschlägigen Internetquellen verwiesen, die Argumente werden in diesem Beispiel nicht verwendet.

Üblicherweise muss ein Device nach dem Öffnen initialisiert werden (Reset). Da hier keine reale Hardware angesprochen wird, kann dieser Schritt entfallen

```
int memory_open(struct inode *inode, struct file *filp) {

  /* Reset Device */

  /* Success */

  return 0;
}
```

Schließen des Moduls als Datei

Um den Zugriff auf das Device zu beenden wird eine Funktion analog zum `fclose` verwendet. In der Struktur `register_chrdev` wird dazu die Funktion close mit der Funktion `memory_release` initialisiert.

Beim Schließen einer Datei sind eventuell Speicherbereiche freizugegeben, die beim Öffnen angefordert wurden. In diesem Beispiel ist das nicht nötig.

```
int memory_release(struct inode *inode, struct file *filp) {
```

```
  /* release memory allocated for opening the device */

  /* Success */
  return 0;
}
```

Daten vom Modul auslesen

Um vom Treiber zu lesen (analog zu `fread`), wird die Funktion `read` verwendet. In dem vorliegenden Beispiel wird dazu die Funktion `memory_read` ausgeführt. Die Argumente sind eine File-Struktur:

- der Buffer `buf` (der mit `fread` aus dem Userspace ausgelesen wird)
- ein Zähler, der die Anzahl der zu übertragenen Bytes angibt (`count`)
- eine Positionsangabe `f_pos`, ab der begonnen werden soll zu lesen

In diesem Beispiel wird lediglich ein Byte aus dem Buffer des Kernel Moduls in den User Space kopiert. Dazu wird die Funktion `copy_to_user` verwendet.

```
ssize_t memory_read(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos) {

  /* Transfering data to user space */
  copy_to_user(buf,memory_buffer,1);

  /* Changing reading position as best suits */
  if (*f_pos == 0) {
    *f_pos+=1;
    return 1;
  } else {
    return 0;
  }
}
```

Zu beachten ist, dass die Leseposition (`f_pos`) in der Datei ebenfalls geändert wird. Ist die Position auf den Beginn der Datei gesetzt, wird die Position um eins erhöht und die Anzahl der Bytes, die gelesen wurde wird als Rückgabeparameter übergeben (hier also 1). Wenn die Position nicht auf dem Beginn der Datei steht, wird eine 0 (end of file) zurückgegeben, da nur ein Byte gespeichert werden soll.

Daten an das Modul schreiben

Um Daten zum Treiber zu übertragen wird Funktion `write` verwendet. In unserem Modul dient dazu die Funktion `memory_write`. Als Parameter dienen

- eine `file` Struktur
- ein Buffer (`buf`), in den mit der User Space Funktion `frwite` geschrieben wird
- ein Zähler `count`, der die Anzahl der zu übertragenden Bytes angibt
- die Position `f_pos`, die angibt an welcher Stelle in die Datei geschrieben werden soll

```
ssize_t memory_write( struct file *filp, char *buf,
                      size_t count, loff_t *f_pos) {

  char *tmp;

  tmp=buf+count-1;
  copy_from_user(memory_buffer,tmp,1);
  return 1;
}
```

In diesem Fall werden die Daten mit `copy_from_user` aus dem User in den Kernel Space übertragen.

Wenn die obigen Codeabschnitte in einer Datei zusmmengefügt werden, kann das Modul kompiliert und wie im nächsten Abschnitt beschrieben getestet werden.

Verwendung des Moduls

```
# sudo insmod memory.ko
```

Die Gerätedatei muss mit den nötigen Rechten versehen werden, um auf sie zugreifen zu können

```
# sudo chmod 666 /dev/memory
```

Anschließend kann in das Device-File `/dev/memory` ein String geschrieben werden. Der letzte char dieses Strings wird dort abgespeichert. Beispiel:

```
$ echo -n abcdef >/dev/memory
```

Der Inhalt des Devices kann mit dem `cat`-Befehl ausgelesen werden:

```
$ cat /dev/memory
```

Quellen:

Die Aufgabenstellung basiert teilweise auf folgendem Artikel, in dem auch weiterführende Anwendungsmöglichkeiten vorgestellt werden:
http://www.freesoftwaremagazine.com/articles/drivers_linux