Linux Kernel Driver Development

Musterlösungen

Roger Knecht
(knechro1@students.zhaw.ch)

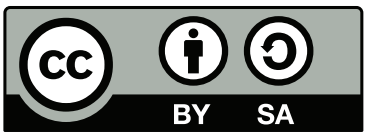
8. März 2015

Semesterarbeit

im 6. Semester

Dozent: Beat Seeliger

Zürcher Hochschule
für Angewandte Wissenschaften



Inhaltsverzeichnis

1 Musterlösung Grundlagen

1.1 Minimaler Kernel bauen

Als erster Schritt soll der Kernel selber kompiliert werden. Versuchen Sie ein möglichst kleines Image zu erstellen, indem Sie Optionen abschalten. Erstellen Sie zunächst eine Konfiguration über *Menuconfig* und messen Sie anschliessend die Dauer des Kompilierens und die Grösse des Images.

```
$ cd linux-source
$ make menuconfig
```

Die Navigation erfolgt über die Pfeiltasten. Mit der Leertasten können Optionen an- bzw. abgewählt werden und mittels der Tabulatortaste kann der Fokus verändert werden.

Kompilieren Sie nun den Kernel und messen sie mit *time* die Dauer. Schreiben Sie alle Zeiten auf (*real*, *user*, *sys*). Mit dem Unix-Tool *du* (Disk usage) kann die Grösse des Images gemessen werden.

```
$ time make
make[1]: Nothing to be done for 'all'.
  HOSTCC  scripts/basic/fixdep
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
[...]
real   XXmX.XXXs
user   XXmX.XXXs
sys    XXmX.XXXs

$ du -h arch/x86/boot/bzImage
X.XM   arch/x86/boot/bzImage
```

Ergebnisse eintragen:

real ~10min bzImage <= 10M
user user + sys <= real
sys __m__.___s

Wie könnte die Imagegrösse weiter verkleinert werden bzw. was hat alles einen Einfluss?
Befehlssatz der CPU, Kompressionsverfahren, Module dynamisch/statisch laden

1.2 Kernel-Modul erstellen

Erstellen Sie nun ein eigenes Kernel-Modul. Legen Sie dafür die Ornderstruktur wie in Abbildung 1 an.

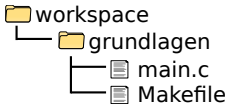


Abbildung 1: Ordnerstruktur

Nehmen Sie dazu das Codebeispiel aus der Vorlesung und kopieren Sie dieses in das *main.c* und füllen Sie das Makefile mit folgendem Inhalt:

Listing 1: Makefile

```
MODULE_NAME = grundlagen
SRC := main.c

KDIR := /lib/modules/$(shell uname -r)/build

obj-m := $(MODULE_NAME).o
$(MODULE_NAME)-objs = $(SRC:.c=.o)

PWD := $(shell pwd)

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

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

Kompilieren Sie nun das Modul.

```
$ make
```

Laden Sie das Modul mit *insmod* (Install module).

```
$ sudo insmod grundlagen.ko
```

Entladen Sie das Modul mit *rmmod* (Remove module).

```
$ sudo rmmod grundlagen.ko
```

Die Ausgaben von *printk()* werden ins Kernellog geschrieben. Sie können das Kernellog mit *dmesg* (Dump messages) anzeigen lassen.

```
$ dmesg | tail -n 10
```

Was steht im Kernellog?

[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] div.
[00000.000000] hello world
[00000.000000] good bye world

KERN_DEBUG	Debuggen (Alle Nachrichten)
KERN_INFO	Informative Nachricht
KERN_NOTICE	Wichtige Information
KERN_WARNING	Warnung
KERN_ERR	Fehler
KERN_CRIT	Kritischer Fehler
KERN_ALERT	Fehlerbehebung muss sofort erfolgen
KERN_EMERG	System ist nicht mehr benutzbar

Tabelle 1: Log-Levels

1.3 Log-Level

Ersetzen Sie das Log-Level in *printk()* durch diese in der Tabelle 1.
Wie verhalten sich die Log-Levels? Vergleichen Sie dabei die Ausgabe im Kernellog.
Debug, Info, Notice werden im Kernellog ausgegeben und Warning, Error, Alert und Emergency direkt in der Konsole.

2 Musterlösung Bootprozess

2.1 Printk während des Booten

Der Bootprozess, welcher in der Vorlesung besprochen wurde, soll nun experimentel überprüft werden. Dabei sollen mit Hilfe von *printk()* Einträge ins Kernellog gemacht werden.

Erstellen Sie zunächst einen neuen Ordner *boot* unter *workspace*

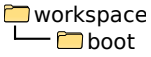


Abbildung 2: Ordnerstruktur für die Aufgabe 2

Entpacken Sie mit den folgenden Befehlen den Linux-Kernel.

```
$ cd ~/workspace/boot
$ cp /usr/src/linux-source-*.tar.bz2 .
$ tar xjf linux-source-*.tar.bz2
```

Modifizieren Sie nun die Funktion *start_kernel()* wie in Listing 2 zu sehen ist, indem Sie die *printk()*-Statements hinzufügen.

Listing 2: init/main.c

```
// [..]

asmlinkage void __init start_kernel(void)
{
        char * command_line;
        extern const struct kernel_param __start___param[], __stop___param[];

        /* modified tux */
        printk(KERN_INFO "#### tux: start_kernel() ####");

        smp_setup_processor_id();

       // [..]
```

Dasselbe machen Sie mit der *rest_init()*...

Listing 3: init/main.c

```
static noinline void __init_refok rest_init(void)
{
        int pid;

        /* modified by tux */
        printk(KERN_INFO "#### tux: rest_init() ####");

        /* [..] */

        /* modified by tux */
        printk(KERN_INFO "#### tux: before cpu_idle()) ####");

        /* Call into cpu_idle with preempt disabled */
        preempt_disable();
        cpu_idle();

        /* modified by tux */
        printk(KERN_INFO "#### tux: after cpu_idle()) ####");
}
```

und der *kernel_init()*-Funktion.

Listing 4: init/main.c

```
static int __init kernel_init(void * unused)
{
        /* modified by tux */
        printk(KERN_INFO "#### tux: kernel_init() ####");

        /*
         * Wait until kthreadd is all set-up.
         */
        wait_for_completion(&kthreadd_done);

        /* [..] */

        init_post();

        /* modified by tux */
        printk(KERN_INFO "#### tux: after kernel_init() ####");

        return 0;
}
```

Anschliessend kompilieren Sie den Kernel. Achten Sie auf Kompilierungsfehler und beheben Sie diese bei Bedarf.

```
$ cd ~/workspace/boot/linux-source-*
$ make
```

Danach installiern Sie den Kernel mit folgendem Befehl:

```
$ make modules_install install
```

Starten Sie die virtuelle Maschine neu und wählen Sie im Grub-Menü (Abbildung 3) den neuen Kernel aus.

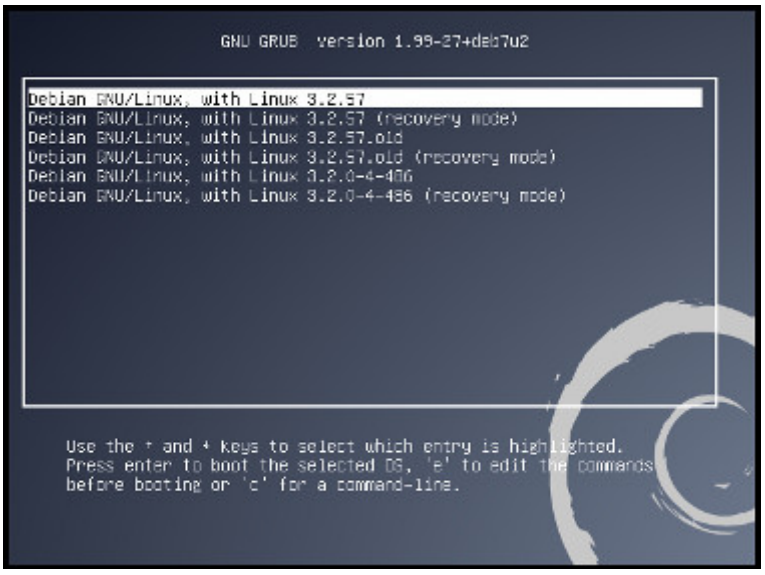


Abbildung 3: Auswahl im Grub-Menu

Analysieren Sie nach dem Boot das Kernellog mit *dmesg*.

```
$ dmesg | less
```

Was stellen Sie dabei fest?

Es werden nur 4 von 5 printk() Ausgaben angezeigt.

Kopieren Sie die Ausgabe des folgenden Befehls in die folgenden Zeilen.

```
$ dmesg | grep "#### tux"
```

[00000.000000] #### tux: start_kernel() ####
[00000.000000] #### tux: rest_init() ####
[00000.000000] #### tux: kernel_init() ####
[00000.000000] #### tux: before cpu_idle() ####
[00000.000000]
[00000.000000]
[00000.000000]

2.2 Init-Programm

Im nächsten Schritt wollen wir das Init-Programm auswechseln. Hierzu müssen wir die Bootparameter von Linux anpassen. Die Bootparameter müssen im Bootloader konfiguriert werden. Editieren Sie hierfür mit Root-Rechten die Datei */boot/grub/grub.cfg*.

Gehen Sie wie folgt vor:

- Suchen Sie in der Datei nach *menuentry*.
- Kopieren Sie den gesammten ersten Menuentry.
- Ändern Sie den Namen zu *Boot to bash*.
- Fügen Sie den Bootparameter *init=/bin/bash* hinzu.

Der neue Eintrag sollte ähnlich des Listing 5 aussehen.

Listing 5: /boot/grub/grub.cfg

```
menuentry 'Boot to bash' --class debian --class gnu-linux --class gnu --class os
    {
        load_video
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='(/dev/sda,msdos1)'
        search --no-floppy --fs-uuid --set=root a2a446aa-571f-4041-b6ce-
            d5f62f336ba5
        echo    'Loading Linux 3.2.57 ...'
        linux   /boot/vmlinuz-3.2.57 root=UUID=a2a446aa-571f-4041-b6ce-
            d5f62f336ba5 ro  quiet init=/bin/bash
        echo    'Loading initial ramdisk ...'
        initrd  /boot/initrd.img-3.2.57
}
```

Speicheren Sie die Datei und starten Sie die virtuelle Maschine neu. Wählen Sie im Grub-Menu den neuen Eintrag *Boot to bash* aus.

Was ist passiert?

Das System bootet direkt in die Bash-Shell

Was sehen Sie bei der Eingabe des Befehls von *whoami*?

Ausgabe: *root*. Das Init-Programm wird immer als Root ausgeführt.

3 Musterlösung VFS

3.1 Eintrag im *procfs*

In folgenden Übung soll eine virtuelle Datei im procfs erstellt werden. Entwickeln Sie hierfür ein Kernel-Modul, welches einen Eintrag unter */proc/hello* erzeugt.

1. Übernehmen Sie im ersten Schritt das *main.c* und *Makefile* von der *Fallstudie Grundlagen*.
2. Ändern Sie den Namen im Makefile zu *vfs_proc*.
3. Fügen Sie die Headerdatei für das *procfs* hinzu:

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

3.1.1 Datei erstellen

Hier sind die zwei für uns relevanten Funktionen des *procfs*:

```
proc_dir_entry* proc_create(
         const char *name,
         mode_t mode,
         struct proc_dir_entry *parent,
         const struct file_operations *proc_fops)

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

Die erste Funktion fügt eine Datei hinzu und die zweite entfernen den Eintrag. *name* entspricht dem Dateinamen, *mode* definiert die Zugriffsrechte, *parent* zeigt auf den Eintrag an dem die Datei angehängt werden soll und mit *fops* kann man die Dateioperationen angeben.

Wir rufen nun die *proc_create()* während der Modulinitialisierung auf und löschen den Eintrag in der Exit-Funktion.

```
static int __init vfs_proc_init (void)
{
        hello_file = proc_create(NAME, MODE, NULL, &hello_fops);
        if (!hello_file) {
                return -ENOMEM; // not enough memory
        }

        printk(KERN_INFO "vfs_proc: init\n");
        return 0;
}

static void __exit vfs_proc_exit (void)
{
        if (hello_file) {
                remove_proc_entry(NAME, NULL);
        }

        printk(KERN_INFO "vfs_proc: exit\n");
}

module_init(vfs_proc_init);
module_exit(vfs_proc_exit);
```

Wenn wir *NULL* bei *parent* übergeben, wird die Datei direkt unter */proc* erzeugt. *NAME* und *MODE* definieren wir über *#define*. Der Pointer *hello_file* ist der Dateieintrag.

```
#define NAME    "hello"
#define MODE    0666    // read+write access to user, group, others
```

```
static struct proc_dir_entry *hello_file;
```

Als nächstes müssen wir die Dateioperationen *hello_fops* definieren. Der Linux-Kernel macht das über eine Struktur namens *file_operations*, welche eine Reihe von *Callback*-Funktionen beinhaltet. Der Struktur übergeben wir unsere eigenen Callback-Funktionen: *hello_open()*, *hello_release()*, *hello_read()* und *hello_write()*. Kopieren Sie diese Struktur **vor** die Init-Funtkion.

```
// callbacks for file operations
static const struct file_operations hello_fops = {
        .owner    = THIS_MODULE,      // pointer to module
        .open     = hello_open,       // callback for open()
        .release  = hello_release,    // callback for close()
        .read     = hello_read,       // callback for read()
        .write    = hello_write,      // callback for write()
};
```

Die Callback-Funktionen belassen wir zunächst sehr einfach:

```
// gets called on file open
static int hello_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "hello: open\n");
        return 0;
}

// gets called on file close
static int hello_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "hello: close\n");
        return 0;
}

// gets called on file read
static ssize_t hello_read(struct file* file, char __user* buf, size_t buf_size,
    loff_t* offset)
{
        printk(KERN_INFO "hello: read\n");
        return 0;
}

// gets called on file write
static ssize_t hello_write(struct file* file, const char __user* buf, size_t
    buf_size, loff_t* offset)
{
        printk(KERN_INFO "hello: write\n");
        return buf_size;
}
```

Speichern Sie nun die *main.c* Datei, kompilieren Sie das Programm über *make* und laden Sie das Modul über *insmod*:

```
$ make
make -C /lib/modules/3.2.57/build M=/home/tux/workspace/vfs/procfs modules
make[1]: Entering directory
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory
$ sudo insmod vfs_proc.ko
```

Mit *rmmod* können Sie das Modul wieder entladen.

```
$ rmmod vfs_proc
```

Was passiert bei *ls -l /proc/hello*?

-rw-rw-rw- 1 root root 0 Aug 18 21:49 /proc/hello

Was passiert bei *cat /proc/hello*?

Dateiinhalt ist leer.

Was passiert bei *echo bye > /proc/hello*?

Schreiboperation beendet ohne Fehler, aber der Dateiinhalt verändert sich nicht.

Was steht anschliessend im Kernellog?

[00000.000000] vfs_procfs: init
[00000.000000] hello: open
[00000.000000] hello: read
[00000.000000] hello: close
[00000.000000] hello: open
[00000.000000] hello: write
[00000.000000] hello: close
[00000.000000] vfs_proc: exit

3.1.2 Dateioperationen

Um den Inhalt der Datei zu definieren, müssen wir die *hello_read()* Funktion erweitern.

```
static ssize_t hello_read(struct file* file, char __user* buf, size_t buf_size,
    loff_t* offset)
{
        // determine length of CONTENT (if buffer size < content length, then
            use buf_size instead)
        size_t size = min(buf_size, strlen(CONTENT));

        // if file offset is > 0, then return with 0 bytes read
        if (*offset > 0) {
                return 0;
        }

        // write CONTENT into userspace memory
        if (copy_to_user(buf, CONTENT, size) != 0) {
                return -EFAULT;
        }

        // increment offset
        *offset += size;

        printk(KERN_INFO "hello: read\n");
        return size;
}
```

Zuerst ermitteln wir die Anzahl Bytes, die wir in den Userspace schreiben. Danach überprüfen wir, mit welchen Offsets die Datei gelesen wird. Falls nicht der Anfang der Datei gelesen wird, melden wir dem Kernel das Dateiende (return 0). Würden wir das nicht machen, würde der Kernel die Datei unendlich weiterlesen, weil nie ein Dateiende kommt (Probieren Sie es aus!). Der Kern unser Funktion ist der *copy_to_user()* Aufruf. Weil wir im Kernelspace arbeiten

können wir nicht direkt auf den Buffer *buf* zugreifen. *copy_to_user()* kopiert die Anzahl Bytes von unserem String *CONTENT* in den Buffer *buf*. Danach inkrementieren wird die Offsetvariable und geben die Anzahl geschriebenen Bytes zurück.

CONTENT definieren wir als statischen String:

```
#define CONTENT "hello reader!\n"
```

Kompilieren und laden Sie das Modul. Vergessen Sie nicht das alte Modul zuvor mit *rmmod* wieder zu entfernen.

Was passiert nun bei *cat /proc/hello*?

Ausgabe: hello reader!

3.1.3 Zusatzaufgabe (Optional)

Erweitern Sie den Code so, dass die Datei beschrieben werden kann und beim anschlissenden Lesen wird der neue Inhalt ausgegeben. Beispiel:

```
$ cat /proc/hello
hello! hello!
$ echo bye > /proc/hello
$ cat /proc/hello
bye
$
```

Verwenden Sie dazu analog zu *copy_to_user()* die folgende Funktion, um den Buffer in *hello_write()* zu lesen:

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)
```

Schreiben Sie den Bufferinhalt in eine globale Variable. Beispiel:

```
static char content[200];
```

4 Musterlösung System Call Interface

4.1 System Call erstellen

In dieser Übung wollen wir nun einen eignen System Call entwickeln. Erstellen Sie einen neuen Ordner im Workspace mit dem Namen *syscall*. Entpacken Sie den Kernel wie bereits in der *Fallstudie Bootprozess* und legen Sie diesen unter *workspace/syscall/linux-source* ab.

Als nächstes fügen wir unseren System Call in der System Call Table ein. Editieren Sie hierzu die Datei *arch/x86/kernel/syscall_table_32.S* wie im Listing 6, indem Sie *.long sys_hello* hinzufügen.

Listing 6: arch/x86/kernel/syscall_table_32.S

```
        // ..

        .long sys_sendmmsg              /* 345 */
        .long sys_setns
        .long sys_process_vm_readv
        .long sys_process_vm_writev

        // hello world system call (nr. 349)
        .long sys_hello

        // ..
```

Jeder System Call hat eine eigene Nummer, die beim Aufruf über das *%eax* Register übertragen wird. Diese Nummer teilen Sie dem Kernel in der nächsten Datei mit (Listing 7). Auch die Anzahl der System Calls *NR_syscalls* muss erhöht werden.

Listing 7: arch/x86/include/asm/unistd_32.S

```
// ..
#define __NR_process_vm_readv   347
#define __NR_process_vm_writev  348
#define __NR_hello              349

#ifdef __KERNEL__

#define NR_syscalls 350

// ..
```

Der Funktionsprototyp wird unter *include/linux/syscall.h* definiert.

Listing 8: include/linux/syscall.h

```
// ..
asmlinkage long sys_process_vm_writev(pid_t pid,
                                      const struct iovec __user *lvec,
                                      unsigned long liovcnt,
                                      const struct iovec __user *rvec,
                                      unsigned long riovcnt,
                                      unsigned long flags);
asmlinkage long sys_hello(void);

// ..
```

Nun müssen wir eine neue Datei erstellen, um die Funktion zu implementieren. Erstellen Sie folgende Datei: *kernel/hello.c* und kopieren Sie den Inhalt von Listing 9 hinein.

Listing 9: kernel/hello.c

```
#include <linux/kernel.h>
#include <linux/syscalls.h>

asmlinkage long sys_hello(void)
{
        printk(KERN_INFO "Hello system call!\n");
        return 0;
}
```

Damit die neue Datei in den Kernel kompiliert wird, müssen wird diese im Makefile angeben (Listing 10). Dafür muss nur *hello.o* am Ende der *obj-y* Variable angegeben werden.

Listing 10: kernel/Makefile

```
#
# Makefile for the linux kernel.
#

obj-y     = sched.o fork.o exec_domain.o panic.o printk.o \
            cpu.o exit.o itimer.o time.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o \
            rcupdate.o extable.o params.o posix-timers.o \
            kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
            hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
            notifier.o ksysfs.o sched_clock.o cred.o \
            async.o range.o hello.o

# ..
```

Der System Call ist nun implementiert. Um diesen zu testen, müssen wir nachfolgenden den Kernel kompilieren, installieren und neustarten.

```
$ make
$ sudo make install
```

4.2 System Call aufrufen

Im zweiten Schritt wird der System Call durch ein Userspace Programm aufgerufen. Legen Sie einen neuen Ordner *syscall_test* unter *workspace/syscall* an. Erstellen Sie eine neue Datei *main.c*.

Wie wir in der Vorlesung gesehen haben, wird ein System Call per Interrupt ausgelöst. Anstatt den Aufruf in Assembly zu schreiben, bietet die *libc* die Funktion *syscall()* an. Kopieren Sie den Code aus Listing 11 in die Maindatei.

Listing 11: workspace/syscall/main.c

```
#include <stdio.h>
#include <sys/syscall.h>

#define SYSCALL_HELLO 349

int main(int argc, char *argv[])
{
        int ret;
```

```
        printf("testing system call...\n");
        ret = syscall(SYSCALL_HELLO);
        printf("executed with return value %i\n", ret);

        return 0;
}
```

Um das Programm zu kompilieren, legen wir noch ein Makefile an:

Listing 12: workspace/syscall/Makefile

```
all:
        gcc -osyscall_test main.c

clean:
        rm -f *.o
```

Kompilieren Sie nun das Programm und führen Sie es aus.

```
$ cd ~/workspace/syscall/syscall_test
$ make
$ ./syscall_test
```

Was ist die Ausgabe?

testing system call...
executed with return value 0

Geben Sie die Ausgabe von *dmesg | tail* an:
[00000.000000] _____
[00000.000000] _____
[00000.000000] _____
[00000.000000] _____
[00000.000000] _____
[00000.000000] _____
[00000.000000] Hello system call!

5 Musterlösung Device Driver

Legen Sie für diese Übung einen neuen Ordner*driver* unter dem *workspace* an. Unter *driver* soll für die nächsten zwei Übungen jeweils ein Ordner erstellt werden: *char* und *block*.

5.1 Char Driver

Schauen Sie sich nocheinmal das *Char Driver* Beispiel in der Vorlesung an und vergleichen Sie es mit der */dev/zero* Gerätedatei. Was muss man im Beispiel von der Vorlesung ändern, um den gleichen Output wie die *zero*-Gerätedatei zu erhalten?

Anstelle von "a\n", muss "0" ausgegeben werden

Überprüfen Sie Ihren Vorschlag indem Sie es implementieren. Bringen Sie im ersten Schritt den Treiber von der Vorlesung zum Laufen. Übernehmen Sie das Makefile aus den vorherigen Übungen. Kompilieren und verwenden können Sie das Modul über die bereits bekannten Befehle:

```
$ make
$ sudo insmod mychar.ko
```

Sobald Sie dies geschafft haben, nehmen Sie ihre Anpassungen vor.

5.2 Block Driver

In dieser Übung soll ein Block driver entwickelt werden. Das Listing im Abschnitt Block-Driver in der Vorlesung ist die Ausgangslage. Bringen Sie dieses zunächst zum Laufen. Kopieren Sie auch hier das Makefile ins Verzeichnis und ändern Sie in der Datei den Namen. Builden Sie das Modul mit *make* und starten Sie es mit *insmod*. Wie gross ist die virtuelle Disk? Überprüfen Sie das mit *cfdisk*.

```
$ make
$ sudo insmod memdrive.ko
$ sudo cfdisk /dev/memdrive
```

Vergrösseren Sie nun die virtuelle Disk auf 5 MB. Wie gross müssen Sie *nsectors* wählen? Schreiben Sie die Berechnung und das Ergebnis auf.

5 MB = 5242880 Bytes => nsectors = 5242880 Bytes / 512 Bytes (Block Size) = 10240

Partitionieren Sie nun die Disk über *cfdisk*. Es soll nur eine Partitionen erstellt und die kompletten 5 MB Speicher verwendet werden. Schreiben (*write*) und Beenden Sie anschliessend (*quit*). Es exsitiert nun eine neue Datei mit dem Namen */dev/memdrive0*. Formatieren Sie die Partition mit *ext2*:

```
$ sudo mkfs.ext2 /dev/memdrive0
```

Danach können Sie die Partition mounten:

```
$ sudo mount /dev/memdrive0 /mnt
```

Erstellen Sie nun einige Dateien und Ordner unter */mnt*. Unmounten und mounten Sie das Verzeichnis anschlissend:

```
$ sudo umount /mnt
$ sudo mount /dev/memdrive0 /mnt
```

Was passiert mit den Dateien und Ordnern?
Die Dateien und Ordner sind immer noch vorhanden. (Speicherfreigabe erst bei *memdrive_release()*)

Machen Sie dasselbe nochmals, aber starten Sie auch den Treiber neu:

```
$ sudo umount /mnt
$ sudo rmmod memdrive
$ sudo insmod memdrive.ko
$ sudo mount /dev/memdrive0 /mnt
```

Was passiert jetzt?
Die Datei */dev/memdrive0* exisiert nicht mehr, weil der Speicher freigegeben wurde.