

Projekt: Linux Ramdisk

bearbeitet von:
Andreas Linz & Tobias Sekan

Stand 1. Februar 2013

Inhaltsverzeichnis

1. Einleitung	1
2. Grundlagen	2
2.1. Treiber-Arten	3
3. Die Ramdisk	4
3.1. <code>init</code> , <code>exit</code>	4
3.2. <code>setup_device</code>	6
3.3. Datentransfer (<code>rd_transfer</code> , <code>rd_request</code>)	9
3.4. Festplattengeometrie	10
3.5. Installationsschritte	11
3.5.1. Automatische Einbindung	11
3.5.2. Manuelle Einbindung	12
3.5.3. Entfernen des Moduls	12
3.6. Problembehebung	13
3.7. geplante Verbesserungen	14
3.8. Probleme bei der Implementierung	14
A. Anhang / Quellen	15

1. Einleitung

Im Rahmen des Moduls „Systemprogrammierung“ war ein Linux-Gerätetreiber zu entwickeln. Wir entschieden uns dafür eine Ramdisk; also einen Blockgerätetreiber, zu entwerfen.¹

In der folgenden Dokumentation werden wir einige Grundlagen erklären, auf die notwendigen Begrifflichkeiten eingehen und den erstellten Quellcode analysieren. Außerdem wird die Einbindung und die Nutzung der „Ramdisk“ an einem konkreten Beispiel demonstriert.

Als Entwicklungs- und Testsystem wurde ein 64-Bit Linux Mint mit Kernel-Release **3.5.0-22-generic**² genutzt.

¹Die Begriffe *Ramdisk* und *Ram-Drive* sind gleichbedeutend.

²Das verwendete Kernel-Release kann mit `uname -r` abgerufen werden

2. Grundlagen

Ein *Device-Driver* liegt im *Kernel-Space* und besitzt einen Betriebssystem-, sowie einen Hardwarespezifischen Teil. Damit der Programmierer einer (*User-Space*) Anwendung sich keine Gedanken um die Implementierung der Hardware machen muss, besitzt das Betriebssystem eine Schnittstelle für Systemrufe. Wird in einer *User-Space* Anwendung bspw. `fread()` aufgerufen, dann delegiert das Betriebssystem die Anfrage an den jeweiligen Treiber; z.B. den Festplatten-Treiber, abhängig von dem *[Special-]File* auf das sich der Aufruf bezieht.

Der Aufbau dieser Abstraktionsschichten ist in Abbildung 1 sehr anschaulich dargestellt.

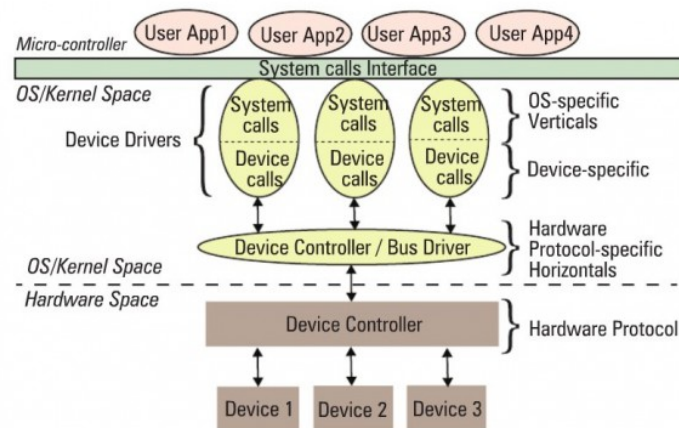


Abbildung 1: Linux Device Driver Partition, [4]

Durch die Aufteilung in Geräte- und Betriebssystemspezifischen Teil des Treibers, genügt es bei der Portierung zu einem anderen Betriebssystem oft nur den letzteres anzupassen.

2.1. Treiber-Arten

Linux kennt grundsätzlich nur drei verschiedene Treiberarten. Diese wären:

Zeichengeräte-Treiber diese Geräteklasse erlaubt nur sequentiellen Zugriff. Es darf immer nur ein Byte pro Anfrage gelesen, oder geschrieben werden. Zu den Zeichengeräten zählt bspw. serielle Ports, Drucker, Soundkarten ...

Blockgeräte-Treiber Blockgeräte bieten, im Gegensatz zu Zeichengeräten, einen wahlfreien Zugriff und erlauben das Transferieren von ganzen Blöcken.. Ein *Block* besteht hierbei immer aus *n-Bytes*, bei Linux ist die Standardblockgröße 512 Bytes.

Paket-orientierte Treiber zu diesem Bereich gehören vorwiegend Netzwerkgeräte.

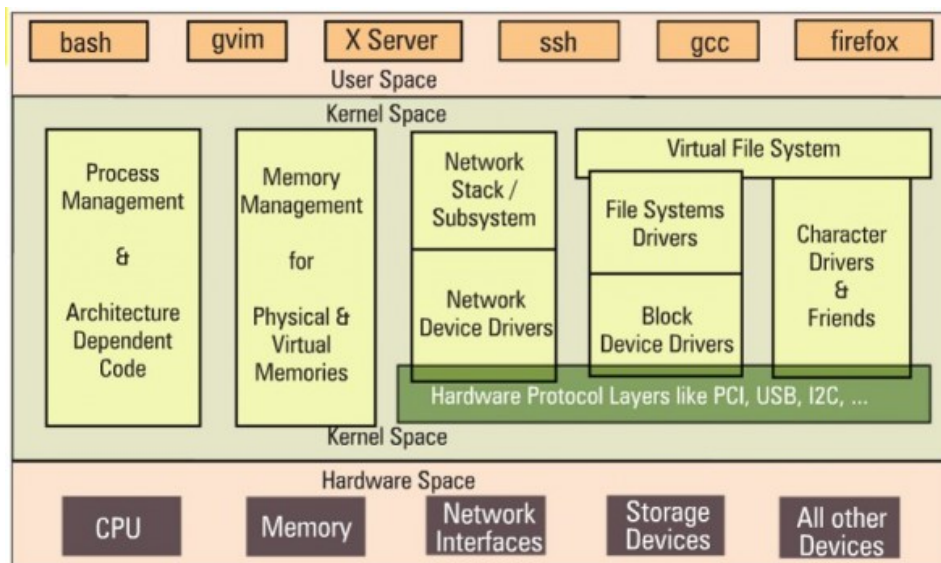


Abbildung 2: Übersicht des Kernelaufbaus, [4]

3. Die Ramdisk

Die erstellte Ramdisk implementiert nur die Funktionen welche zur Nutzung des Blockgerätetreibers nötig sind. Mit anderen Worten stellt sie somit ein Minimalbeispiel für einen solchen Treiber da. Im Gegensatz zum „sbull“ Treiber welcher in *Linux Device Drivers*[1] enthalten ist, wurde auf die Simulation eines Medienwechsels und eines Timeouts verzichtet. Somit entfallen auch die `open`, `release` und `revalidate` Funktionen. Eine kleine Randbemerkung, die `block_device_operation media_change` ist obsolet und muss bei neuen Treibern durch `check_events` ersetzt werden.

Der Treiber besteht aus nur 2 Quelltextdateien, `ramdisk.c` und `ramdisk.h`.

Beginnen will ich mit der Analyse der `init` und `exit` Funktionen des Treibers.

3.1. init, exit

```

1 static int __init ramdisk_init(void)    // constructor
2 {
3     major = register_blkdev(0, DEVICENAME); // fs.h
4     if(major <= 0)
5     {
6         printk(KERN_ALERT PRINT_PREFIX"Could not get a major number
7             !\n");
8         return -EBUSY;
9     }
10    else
11    {
12        printk(KERN_INFO PRINT_PREFIX"Major number %d\n", major);
13        // slab.h, GFP_KERNEL - Allocate normal kernel ram. May
14        sleep.
15        device_arr = kmalloc(DEVICE_COUNT * sizeof(struct
16            ramdisk_dev), GFP_KERNEL);
17        if(!device_arr)
18        {
19            unregister_blkdev(major, DEVICENAME);
20            return -ENOMEM;
21        }
22        else
23        {
24            for(int i = 0; i < DEVICE_COUNT; i++)
25            {
26                struct ramdisk_dev* rd;
27                rd = device_arr + i;
28                printk(KERN_INFO PRINT_PREFIX"##d registered\n", i);
29                setup_device(rd, i);

```

3. Die Ramdisk

```
27     }
28     }
29 }
30 return 0;
31 }
```

init-Funktion `ramdisk_init`

Der Code sollte weitestgehend selbst erklärend sein. Die `init` Funktion wird aufgerufen wenn das Kernel-Modul mit `insmod` oder `modprobe`³ geladen wird. `exit` dementsprechend wenn das Modul mit `rmmmod` entfernt wird. Im header-file *ramdisk.h* kann anhand von `DEVICE_COUNT` die Anzahl der zu erstellenden Geräte festgelegt werden. Aus diesem Grund wird in Zeile 13 mit `kmalloc` ein Array von `ramdisk_dev` Strukturen allokiert.

```
1 struct ramdisk_dev
2 {
3     int    sectors;           // # of sectors
4     long   size;              // size in Bytes
5     u8*    data;              // disk data, unsigned char = u8
6     spinlock_t lock;          // Mutex
7     struct request_queue* queue; // device request queue
8     struct gendisk *gd;        // Note: can't be allocated
                                // directly, instead use alloc_disk(int minors), see genhd.h
9 };
```

struct `ramdisk_dev`

In der Schleife (Zeilen 21-27 der `init`-Funktion) werden die einzelnen Geräte durch den Aufruf von `setup_device(rd, i);` (siehe 3.2, S.6) eingerichtet und die `struct ramdisk_dev` Elemente gefüllt.

```
1 static void __exit ramdisk_exit(void) // destructor
2 {
3     for(int i=0; i < DEVICE_COUNT; i++)
4     {
5         struct ramdisk_dev* rd = device_arr + i;
6
7         if(rd->gd)
8             del_gendisk(rd->gd);
9         if(rd->queue)
10            blk_cleanup_queue(rd->queue);
11         if(rd->data) // if allocated, deallocate disk space
12            vfree(rd->data);
13     }
14 }
```

³im Gegensatz zu `insmod` kann `modprobe` (rekursive) Abhängigkeiten auflösen.

3. Die Ramdisk

```
15     unregister_blkdev(major, DEVICENAME); // fs.h
16     kfree(device_arr);
17     printk(KERN_INFO PRINT_PREFIX"unregistered\n");
18
19     return;
20 }
```

exit-Funktion ramdisk_exit

Die Aufgabe der `exit` Funktion besteht im Grunde nur darin den allokierten Speicher freizugeben und die Geräte-Registrierung zu entfernen.

```
1 module_init(ramdisk_init);
2 module_exit(ramdisk_exit);
```

Makros zur Registrierung der `init` und `exit` Funktionen

3.2. setup_device

```
1 static void setup_device(struct ramdisk_dev* rd, int dev_nr)
2 {
3     if(rd == NULL)
4     {
5         printk(KERN_ALERT PRINT_PREFIX"No device with number %d\n",
6             dev_nr);
7     }
8     memset(rd, 0, sizeof (struct ramdisk_dev)); // reset device
9     structure
10
11     if(user_disk_size <= 0)
12     {
13         printk(KERN_INFO PRINT_PREFIX"%d bad size!\nUsing default ->
14             _64MiB", user_disk_size);
15         user_disk_size = 64;
16     }
17     rd->sectors = (user_disk_size * 1024) * NR_OF_SECTR_PER_KB;
18     rd->size     = rd->sectors * SECTOR_SIZE;
19     rd->data      = vmalloc(rd->size);
20     if(rd->data == NULL)
21     {
22         printk(KERN_ALERT PRINT_PREFIX"Could not allocate %d MiB of
23             memory!\n", user_disk_size);
24     }
25     return;
26 }
```


3. Die Ramdisk

```
22     printk(KERN_INFO PRINT_PREFIX"Allocated: %ld Bytes / %d MiB\n",
23            rd->size, (int) rd->size / (0x1 << 20));
24     // RamDisk is a virtual device, but we have to set plausible
25     // disc geometry values
26     rd->geo.heads = 16; // uchar
27     rd->geo.sectors = 32; // uchar
28     rd->geo.start = 0L; // ulong
29     rd->geo.cylinders = rd->size / (SECTOR_SIZE * rd->geo.heads *
30                                   rd->geo.sectors);
31
32     spin_lock_init(&rd->lock); // setup Mutex
33
34     rd->queue = blk_init_queue(rd_request, &rd->lock); // set the
35     // request queue
36     if (rd->queue == NULL)
37     {
38         printk(KERN_ALERT PRINT_PREFIX"Could not allocate request-
39                queue!\n");
40         goto free_and_exit;
41     }
42     blk_queue_logical_block_size(rd->queue, SECTOR_SIZE);
43     rd->queue->queuedata = rd;
44
45     rd->gd = alloc_disk(MINOR_COUNT); // genhd.h
46     if (!rd->gd)
47     {
48         printk(KERN_ALERT PRINT_PREFIX"Disc allocation failure!\n");
49         goto free_and_exit;
50     }
51
52     rd->gd->major = major;
53     rd->gd->first_minor = dev_nr * MINOR_COUNT;
54     rd->gd->fops = &rd_ops;
55     rd->gd->queue = rd->queue;
56     rd->gd->private_data = rd;
57
58     snprintf(rd->gd->disk_name, 32, DEVICENAME"%c", 'a'+dev_nr);
59     // set device name
60     printk(KERN_INFO PRINT_PREFIX"Diskname: %s\n", rd->gd->disk_name);
61     set_capacity(rd->gd, rd->sectors);
62
63     add_disk(rd->gd);
64
65     return;
```

3. Die Ramdisk

```
61
62     free_and_exit :
63         if (rd->data)
64             vfree (rd->data);
65 }
```

setup_device

`setup_device` prüft die Größenangabe `user_disk_size` auf Einhaltung des Wertebereiches, diese Variable ist als Argument bei `insmod` änderbar (siehe 3.5, S.11). Sollte der Test erfolgreich sein, wird versucht mit `vmalloc` die angegebene Diskgröße zu allokieren. Die Größe setzt sich dabei aus der Anzahl der *Sektoren* und der *Sektorgröße* zusammen. Letztere ist im Normal 512 Byte in Linux-Systemen.

Im nächsten Schritt wird die `rd_geo`-Struktur vom Typ `hd_geometry` gefüllt, sie enthält die Festplattegeometriedaten der Ramdisk. Das Gerät ist zwar nur virtuell, aber Programme wie *fdisk* verlangen die Geometriedaten, weshalb an dieser Stelle versucht wird plausible Geometrieangaben zu erstellen.⁴

Der Mutex wird in Zeile 30 initialisiert.

Zeile 32-39 dient zur Festlegung der *strategy*. In diesem Beispiel eine sehr minimalistische Warteschlangenimplementierung in `rd_request`, außerdem wird die Blockgröße des Gerätes dem Kernel in Zeile 38 mitgeteilt. Die Anfragen an die Requestfunktion werden somit vom *Kernel* an die übergebene Sektorgröße angepasst.

Die `generic disk`-Struktur `gd` wird in den Zeilen 48-52 gefüllt. In ihr ist bspw. die Major-Nummer und die erste Minor-Nummer des Gerätes enthalten. Außerdem die `file_operations`-Struktur mit den Referenzen auf die implementierten Blockgerätefunktionen, sowie eine Referenz auf die Warteschlange `queue`.

Im folgenden wird noch der Gerätenamen erstellt, alphabetisch aufsteigend, und die Kapazität in *Sektoren*⁵ festgelegt.

Zum Schluß wird die Disk mit `add_disk` den hinzugefügt (nun in `/dev`) und ist ab dem Moment *live*, d.h. es können sofort Anfragen an das Gerät kommen!

```
1 // blkdev.h
2 static struct block_device_operations rd_ops =
3 {
4     .owner          = THIS_MODULE,
5     .getgeo         = rd_getgeo,
6     .ioctl          = rd_ioctl
7 }
#ifdef MEDIA_CHANGEABLE
```

⁴Einen kurzen Exkurs in die Festplattegeometrie wird in 3.4 auf Seite 10 gegeben.

⁵Häufiger Fehler, die Angabe erfolgt in Sektoren nicht in Bytes! Im *sbull*[1] Beispieldreiber war dies durch ungünstige Variablenbenennung nicht sofort ersichtlich.

3. Die Ramdisk

```
8      ,
9      .open          = rd_open ,
10     .release        = rd_release ,
11     .check_events    = rd_check_events ,
12     .revalidate_disk = rd_revalidate ,
13     #endif // MEDIA_CHANGEABLE
14 };
```

struct block_device_operations

3.3. Datentransfer (rd_transfer, rd_request)

```
1 static void rd_request(struct request_queue* q)
2 {
3     struct request *req;
4
5     req = blk_fetch_request(q);
6     while (req != NULL) {
7         struct ramdisk_dev *rd = req->rq_disk->private_data;
8
9         if (req == NULL || (req->cmd_type != REQ_TYPE_FS)) {
10             printk(KERN_NOTICE PRINT_PREFIX"Skip_non-CMD_request\n");
11             ;
12             __blk_end_request_all(req, -EIO);
13             break;
14         }
15         rd_transfer(rd, blk_rq_pos(req), blk_rq_cur_sectors(req),
16                     req->buffer, rq_data_dir(req));
17         if (!__blk_end_request_cur(req, 0))
18         {
19             req = blk_fetch_request(q);
20         }
21     }
22     return;
```

request-Funktion rd_request

```
1 static void rd_transfer(struct ramdisk_dev* rd, unsigned long sector
2 , unsigned long nsectors, char* buffer, int write)
3 {
4     unsigned long offset = sector * SECTOR_SIZE;
5     unsigned long nbytes = nsectors * SECTOR_SIZE;
```

3. Die Ramdisk

```
5
6  if((offset + nbytes) > rd->size)
7  {
8      printk(KERN_INFO PRINT_PREFIX"Beyond_end_of_disk_(%ld_%ld_-_
          actual_disk_size_%ld)!\n", offset, nbytes, rd->size);
9      return;
10 }
11 if(write)
12     memcpy(rd->data + offset, buffer, nbytes);
13 else
14     memcpy(buffer, rd->data + offset, nbytes);
15 }
```

transfer-Funktion rd_transfer

3.4. Festplattengeometrie

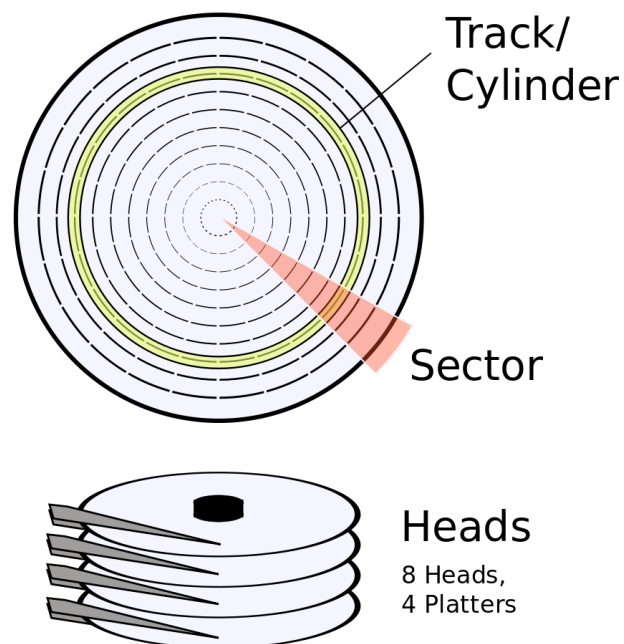


Abbildung 3: Quelle <http://en.wikipedia.org/wiki/Cylinder-head-sector>

Auch wenn die Ramdisk nur ein virtuelles Hardwaregerät ist, muss sie Geometriedaten enthalten, um die Funktion diverser Programme mit ihr zu gewährleisten.

3. Die Ramdisk

Das kleinste Element ist der *Sektor*, die Summe der Sektoren auf einem der konzentrischen Kreise ergibt dann die *Spur* (Track). Die Summe aller Tracks über alle *Heads* ergibt den *Zylinder*, was auf der Grafik leider nicht verdeutlicht ist. `sbull` (siehe [1]) benutzt zur Berechnung der Zylinderanzahl eine etwas eigentümliche Bitmanipulation:

```
1 geo.cylinders = (size & ~0x3f) >> 6;
```

Dies halte ich, gerade weil zu dieser „Bitmagic“ kein Satz im Buch verloren wird, für ein sehr ungünstiges Beispiel. Meine Implementierung sieht folgendermaßen aus:

```
1 rd_geo.heads      = 16;
2 rd_geo.sectors    = 32;
3 rd_geo.start      = 0L;
4 rd_geo.cylinders  = rd->size / (SECTOR_SIZE * rd_geo.heads * rd_geo.sectors);
```

Füllen der Geometriedaten in `ramdisk.c`

Die Größe der Ramdisk lässt sich durch folgende simple Formel nachrechnen:

$$\text{Kapazität} = \text{Sektorgroße} * \text{Sektoranzahl_pro_track} * \text{Zylinderanzahl} * \text{Kopfanzahl}$$

3.5. Installationsschritte

Sollte `ramdisk.ko` noch nicht existieren, ist `make` im Ramdisk-Ordner auszuführen. Der Buildvorgang nutzt `kbuild` (Kernelbuild) und setzt installierte Kernel-Headers im Verzeichnis `/lib/modules/` voraus. Letzteres ist bei dem verwendeten *Linux Mint* vorinstalliert, kann ansonsten aber sehr einfach über `apt` oder ähnliche Paketmanager nachinstalliert werden. Für Debian basierte Distributionen wäre folgendes im Terminal auszuführen:

```
1 apt-get install linux-headers-3.5.0-22-generic
```

3.5.1. Automatische Einbindung

Das Skript `insert_and_test.sh` mit *Superuser*-Rechten starten⁶:

```
1 sudo ./insert_and_test.sh [Disksize in MiB]
```

⁶Sollte die Datei nicht ausführbar sein, dann muss das execute-flag mit `chmod +x insert_and_test.sh` gesetzt werden

3. Die Ramdisk

⁷ Das Skript führt diese Schritte durch:

- Erstellen des mount-Verzeichnisses unter `/media/RamDisk`
- Einfügen des Kernel-Moduls
- Erstellen eines *ext3* Dateisystems auf der Disk
- Mounten der Ramdisk
- Setzen der Berechtigungen für das mount-Verzeichnis
- Ausführung eines kurzen Benchmarks⁸

3.5.2. Manuelle Einbindung

```
1 mkdir /media/RamDisk
2 insmod ramdisk.ko user_disk_size=[Disksize in MiB]
3 sudo mkfs [-t filesystem] /dev/RamDisk_a
4 # Die filesystem Angabe ist optional, standart ist ext2
5 # Wird nur ein Device verwendet, dann ist /dev/RamDisk_a zu nutzen
6 mount -t filesystem /dev/RamDisk_a /media/RamDisk
7 chmod a+wx /media/RamDisk
```

auszuführen als *su* (superuser)

3.5.3. Entfernen des Moduls

```
1 sudo ./remove.sh
```

automatisch

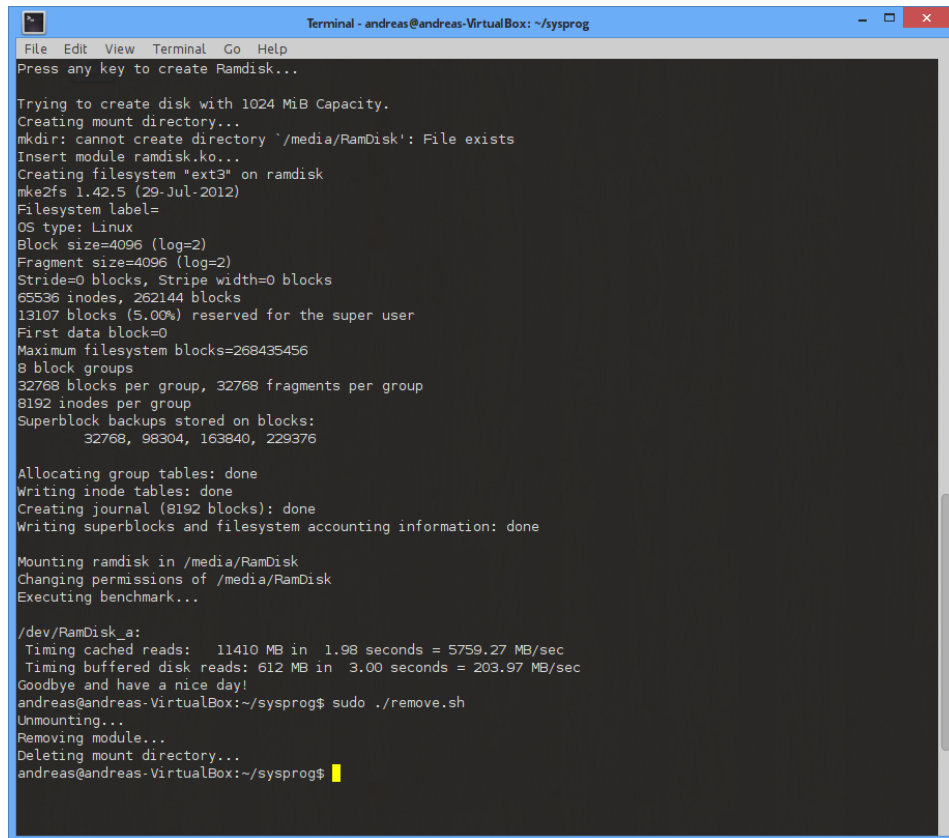
```
1 umount /media/RamDisk
2 rmmod RamDisk
3 rmdir /media/RamDisk
```

Manuelles entfernen (als *su*)

⁷Wird keine Diskgröße angegeben dann verwendet das Modul den Defaultwert von 64 MiB (MiB = 2^{20} Byte). Der Wertebereich des eingegebenen Wertes wird im Skript sowie im Treiber geprüft.

⁸`hdparm -Tt`

3. Die Ramdisk



```
Terminal - andreas@andreas-VirtualBox: ~/sysprog
File Edit View Terminal Go Help
Press any key to create Ramdisk...

Trying to create disk with 1024 MiB Capacity.
Creating mount directory...
mkdir: cannot create directory '/media/RamDisk': File exists
Insert module ramdisk.ko...
Creating filesystem "ext3" on ramdisk
mke2fs 1.42.5 (29-Jul-2012)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
65536 inodes, 262144 blocks
13107 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

Mounting ramdisk in /media/RamDisk
Changing permissions of /media/RamDisk
Executing benchmark...

/dev/RamDisk_a:
Timing cached reads: 11410 MB in 1.98 seconds = 5759.27 MB/sec
Timing buffered disk reads: 612 MB in 3.00 seconds = 203.97 MB/sec
Goodbye and have a nice day!
andreas@andreas-VirtualBox:~/sysprog$ sudo ./remove.sh
Unmounting...
Removing module...
Deleting mount directory...
andreas@andreas-VirtualBox:~/sysprog$
```

Abbildung 4: Terminalausgabe von `insert_and_test.sh` und `remove.sh` unter Xubuntu 12.10

3.6. Problembehebung

`vmalloc` versucht zusammenhängende Speicherbereiche zu allokieren, ist die Angabe der `user_disk_size` zu groß, dann führt das zu einem Fehlschlagen der Speicherallokierung. In Folge dessen kann das Device nicht registriert werden. Um die Fehlermeldung zu sehen muss man den Kernel Ring Buffer auslesen, dazu dient der Befehl `dmesg`. Auf dem Testsystem, welches mit 8GB Arbeitsspeicher ausgestattet ist, wurde die Allokierung von mehr als ≈ 1.5 GiB mit nachfolgendem Fehler quittiert.

```
1 $ dmesg | grep vmalloc
2 [16866.546869] vmalloc: allocation failure: 18446744071562067968
   bytes
3 [16866.546896] [< ffffffff81158d76 >] --vmalloc_node_range+0x1b6/0
   x250
4 [16866.546905] [< ffffffff81158e45 >] --vmalloc_node+0x35/0x40
5 [16866.546912] [< ffffffff81158efc >] vmalloc+0x2c/0x30
```

3. Die Ramdisk

vmalloc Fehlermeldung im Kernel Ring Buffer

Als Verbesserung wäre es möglich die *user_disk_size* sukzessive zu verringern, bis eine erfolgreiche Allokierung möglich ist.

Für die Kompilierung ist außerdem ein aktueller *gcc* notwendig, da `std=gnu11` als im Makefile gesetzt ist. Das heisst, dass der *gcc* mit den aktuellen C-Standard verwenden darf; unter anderem sind nun frei platzierbare Variablendeklaration erlaubt, was der Lesbarkeit des Codes sehr zuträglich ist.

3.7. geplante Verbesserungen

Die einzige, aus Abwärtskompatibilitätsgründen, implementierte *ioctl*-Funktion `HDIO_GETGEO` sollte in eine *unlocked ioctl*-Variante⁹ geändert werden. *Ioctl*-Methoden sollten allgemein sparsam verwendet werden, da sie ein u. U. ein Sicherheitsrisiko darstellen und wenn sie herkömmlich implementiert sind beim Aufruf einen *Big Kernel Lock* (BKL) setzen. Aktuelle Kernel-Releases rufen die in den `block_device_operations` gesetzte `getgeo`-Methode auf, anstatt die Geometrieinformationen per *ioctl*-call zu holen.

3.8. Probleme bei der Implementierung

- Änderungen am Kernel sind scheinbar nicht zentral dokumentiert. Einige Fallstricke sind als Kommentare in den Kernel-Headern hinterlegt, bspw. veraltete Methoden. Die einzige Quelle die das Gros der Modifikation dokumentiert war Linux Weekly News, durch die Verteilung auf einzelne Artikel war aber auch hier die Informationsbeschaffung eher mühselig.
- Alle für uns verfügbaren Fachbücher zu dem Thema bezogen auf sich auf Kernelversionen < 2.6.37.

⁹Switch ioctl functions to → `unlocked_ioctl` (<https://lkml.org/lkml/2008/1/8/213>)

A. Anhang / Quellen

Literatur

- [1] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartmann: „*Linux Device Drivers*“. O'Reilly, 2005. ISBN-13: 978-0596005900
- [2] Javier Martinez: „*Linux Device Drivers 3 examples updated to work in recent kernels*“
<https://github.com/martinezjavier/ldd3/tree/master/sbull>
- [3] Thomas Zink: „*Linux Kernel Module and Device Driver Development*“. Uni Konstanz, https://kops.ub.uni-konstanz.de/xmlui/bitstream/handle/urn:nbn:de:bsz:352-175972/Studienarb-linux_kernel_dev.pdf
- [4] Linuxforu.com: „*Article Series on Linux Device Drivers*“, <http://www.linuxforu.com/tag/linux-device-drivers-series/page/2/>
- [5] Boguslaw Sylla, Patrick Schorn „*Linux Kernel-Module*“, <http://www.cs.hs-rm.de/~weber/sysprog/proj0809/linuxkernel.pdf>