Kernel- und Treiberprogrammierung – Folge 118

Blockabfertigung

Treiber für Blockgeräte, also für SSDs, Festplatten, Bänder oder Flash-Speicher unterscheide[n] sich fundamental von Treibern für zeichenorientierte Geräte. Wir zeigen, wie sie aufgebaut sind und funktionieren. Eva-Katharina Kunst, Jürgen Quade

Anders als bei zeichenorientierten Geräten greifen User auf Blockgeräte nur indirekt zu. Stattdessen avanciert der Kernel zum eigentlichen Auftraggeber. Er arbeitet aus Performance-Gründen meist mit einem Cache (dem Page Cache) und sortiert bei Bedarf einen Reigen von Aufträgen mithilfe des I/O-Schedulers um 1.

Außerdem ermöglicht der Kernel den Zugriff auf einzelne Bytes, obwohl die Geräte Daten grundsätzlich nur in Blöcken verarbeiten – daher ja auch der Name Blockgerät. Ein Block ist typischerweise 512 Byte groß oder ein Vielfaches davon. Ein einzelnes Byte auf ein Blockgerät schreiben? Geht nicht. Man erfindet 511 Bytes dazu und transferiert dann alles. Ein einzelnes Byte lesen? Geht nic[ht.] Man liest 512 Bytes und separiert das gesuchte. Dank Blockgeräte-Subsystem muss sich die Anwenderin oder der An[-]wender glücklicherweise nicht um dies[e] Details kümmern, auch nicht die Entwi[ck]lerin oder der Entwickler, die den Treib[er] für ein Blockgerät realisieren. Der Treib[er] erhält vom Kernel einen Auftrag (Reque[st]) auf dem Blockgerät ab dem Block X ei[ne] sequenziell folgende Anzahl von Blöcke[n]

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade, Professor an der Hochschule Niederrhein, führt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux durch.

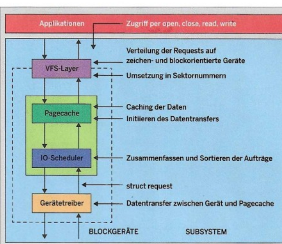
in den Hauptspeicher zu wuchten. Allerdings – und da fangen die Schwierigkeiten an – kann der im Reqeust spezifizierte Hauptspeicherbereich „zerstückelt", also unzusammenhängend sein. Doch dazu später mehr 2.

Zuvor meldet der Treiber nämlich beim Kernel das Blockgerät mit seinen Zugriffsfunktionen an und liefert ausreichend Informationen mit, um vom Kernel korrekt angesprochen zu werden. Dazu sind mehrere Datenstrukturen und Funktionen zu definieren und zu initialisieren 3.

Zentral ist `struct gendisk`. Diese Datenstruktur beschreibt das eigentliche Blockgerät, auf das beziehungsweise von dem Daten transferiert werden. Es repräsentiert den gesamten vom Gerät zur Verfügung gestellten Speicher und abstrahiert die Hardwaredetails. Die auf die Struktur ansetzbaren Methoden listet Tabelle 1. Der Programmierer reserviert per `alloc_disk()` Speicher, initialisiert die Struktur und übergibt diese schließlich per `add_disk()` dem Blockgeräte-Subsystem des Kernels. Sollte später der Treiber wieder entladen werden, gibt er per `del_gendisk()` das Gerät respektive den Speicher frei.

Abstrakte Geräte

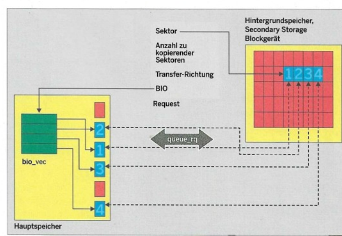
Ein Blockgerät wie eine SSD wird im Wesentlichen durch die folgenden Attribute beschrieben: den Namen (`disk_name`), eine Liste von zu implementierenden Zugriffsfunktionen (`fops`), eine Liste für die Schreib-/Leseaufträge (`queue`), die Größe der SSD (`capacity`) in Sektoren, wobei jeder Sektor 512 Byte groß ist, und schließlich eine Gerätenummer, über die sich der zugehörige Treiber später aus dem Userland identifizieren lässt. Die Gerätenummer besteht aus zwei Teilen, der Treibernummer (`major`) und der Minor-Nummer (`minor`). Die Treibernummer teilt der Kernel bei Aufruf der Funktion `register_blkdev()` zu.



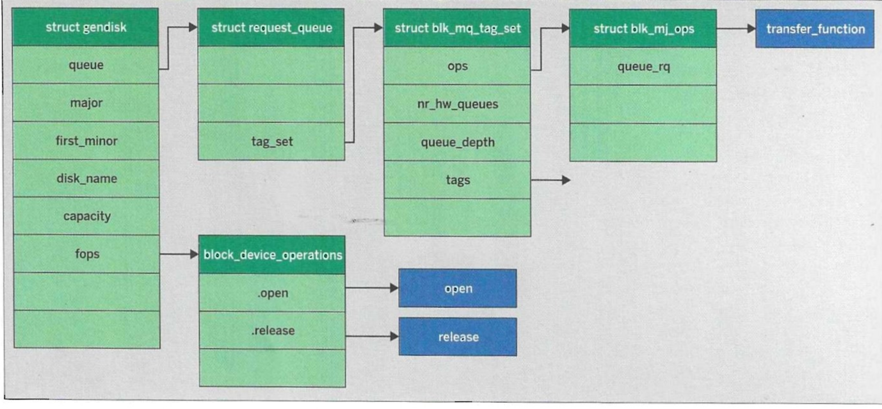
1 Die Architektur des Blockgeräte-Subsystems.

Die Minornummer repräsentiert eine auf der Disk potenziell angelegte Partition; Eine Null steht dabei für die komplette Disk, eine Eins für die erste Parition, eine Zwei für die zweite Parition und so weiter.

In eine Datenstruktur vom Typ `struct block_device_operations` werden die Adressen von Zugriffsfunktionen eingetragen. Das sind zum Beispiel eine open- und eine `release`-Funktion, deren Aufruf



2 Zusammenhängende Daten auf der Disk können im Hauptspeicher zerstückelt liegen.



3 Komponenten, die für einen Blockgerätetreiber anzulegen sind.

erfolgt, wenn das Blockgerät partitioniert werden soll oder ein Filesystem angelegt wird. Die Funktion, deren Adresse unter open abgelegt ist, gibt im einfachsten Fall nur Null zurück, `release` hat keinerlei Rückgabewert. Darüber hinaus lassen sich hier Funktionen einhängen, die beispielsweise die Anfragen bezüglich der Blockgeräte-Geometrie beantworten (`getgeo`) oder aufgerufen werden, wenn beispielsweise das Geräte-Medium geändert wird (`media_changed`).

Die eigentlichen Datentransferaufträge werden als Requests bezeichnet. Jedes Blockgerät definiert mindestens eine HW-Request-Liste, mit der die zu bearbeitenden Requests verwaltet werden. Bei dem in Linux implementierten *Multi-Queue Block Queuing* (block-mq) gibt es zusätzlich pro CPU eine Softwareliste, auch *Staging Queue* genannt, in der die Requests zunächst eingestellt werden. Von dort transferiert sie dann eventuell aus Performance-Gründen der bekannte

Listing 1: Blockgerätetreiber für Ramdisk

```
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/blk-mq.h>

#define NR_SECTORS 204800

static struct gendisk *dummy_disk;
static struct blk_mq_tag_set dummy_tag_set;
static u8 *ram_bd;

static int do_request(struct request *rq, unsigned
int *nr_bytes)
{
  struct bio_vec bvec;

  struct req_iterator
 iter;
  loff_t pos = blk_rq_pos(rq) << SECTOR_SHIFT;
  loff_t dev_size = (loff_t)(NR_SECTORS*512);
  unsigned long b_len;
  void *b_buf;

  printk("dummy_bd: request start from sector %lld
"
    "pos = %lld  dev_size = %lld\n",
    blk_rq_pos(rq), pos, dev_size);

  rq_for_each_segment(bvec, rq, iter) {
    b_len = bvec.bv_len;
    b_buf = page_address(bvec.bv_page) + bvec.bv_
offset;

    // out of memory bounds?
    if ((pos + b_len) > dev_size) {
      printk("do_request: out of memory
bounds()\n");
      b_len = (unsigned long)(dev_size - pos);
    }
```

Listing 1: Blockgerätetreiber für Ramdisk (Fortsetzung)

```
    if (rq_data_dir(rq) == WRITE) {
      memcpy(ram_bd + pos, b_buf, b_len);
    } else {
      memcpy(b_buf, ram_bd + pos, b_len);
    }
    pos += b_len;
    *nr_bytes += b_len;
  }
  return 0;
}

static blk_status_t dummy_block_request(
  struct blk_mq_hw_ctx *hctx,
  const struct blk_mq_queue_data *bd )
{
  struct request *req;
  blk_status_t status = BLK_STS_OK;
  unsigned int nr_bytes = 0;

  req = bd->rq;
  blk_mq_start_request( req );

  if (blk_rq_is_passthrough( req )) {
    blk_mq_end_request( req, BLK_STS_IOERR );
    return status;
  }
  printk("transfer %s: from %llu count: %u\n",
    rq_data_dir( req ) ? "write" : "read",
    blk_rq_pos( req ), blk_rq_bytes( req) );
  // do the transfer
  if (do_request(req, &nr_bytes) != 0) {
    status = BLK_STS_IOERR;
  }
  // Notify kernel about processed nr_bytes
  if (blk_update_request(req, status, nr_bytes)) {
    BUG();
  }
  blk_mq_end_request( req, status );
  return status;
}

static int bd_open( struct block_device *bdev,
fmode_t mode )
{
  return 0;
}

static void bd_release( struct gendisk *gd, fmode_t
mode )
{
  return;
}

static struct blk_mq_ops dummy_queue_ops = {
  .queue_rq = dummy_block_request,
};

static struct block_device_operations bd_ops = {
  .owner = THIS_MODULE,
  .open = bd_open,
  .release = bd_release,
};

static int __init dummy_bd_init(void)
{
  int err;

  // allocate memory for device data
  ram_bd = vmalloc( NR_SECTORS*512 );
  if ( ram_bd==NULL ) {
    printk("vmalloc failed\n");
    return -ENOMEM;
  }

  // prepare tag_set
  dummy_tag_set.ops = &dummy_queue_ops; // add
transfer-function
  dummy_tag_set.nr_hw_queues = 1;
  dummy_tag_set.queue_depth = 64;
  dummy_tag_set.numa_node = NUMA_NO_NODE;
  dummy_tag_set.cmd_size = 0;
  dummy_tag_set.flags = BLK_MQ_F_SHOULD_MERGE;
  err = blk_mq_alloc_tag_set( &dummy_tag_set );
  if (err)
    goto fail_alloc_tag_set;

  // prepare gendisk and bind it together
  dummy_disk = alloc_disk( 3 ); // allow two
partitions
  if (dummy_disk == NULL) {
    printk("alloc_disk failed\n");
    goto fail_alloc_disk;
  }
```

I/O-Scheduler umsortiert in die Hardware-Queue. Diese Architektur reduziert die sonst notwendigen Locking-Mechanismen und erhöht dank der Parallelität die Gesamtperformance.

Alter Verwalter

Die Verwaltung der Requests selbst erfolgt über sogenannte Tag-Sets. Jedes Tag-Set besteht dabei aus einer oder mehreren Tag-Gruppen, jede Tag-Gruppe wiederum aus einer Anzahl von Tags sowie den eigentlichen Requests, die der Tiefe der zum Blockgerät gehörenden HW-Request-Queue entspricht. Pro HW-Request-Queue gibt es eine Tag-Gruppe. Soll also ein Auftrag an das Blockgerät eingereiht werden, dann reserviert der Kernel zunächst einen Tag. Ist dies erfolgreich, reiht er den Request ein, sonst ist erst einmal Warten angesagt.

Sowohl die Software- als auch die Hardware-Queues werden über die Funktionen `blk_mq_init_queue()` indirekt erzeugt beziehungsweise über `blk_cleanup_queue()` wieder freigegeben. Die Funktionen verbinden die `struct gendisk` über die Request-Queue mit dem im Treiber zu definierenden Tag-Set und reservieren entsprechend der Parametrierung notwendigen Speicher.

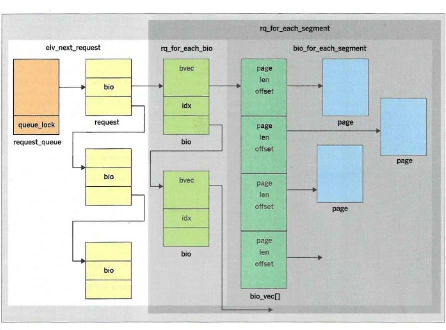
Unter anderem werden in dieser Struktur die Anzahl der Hardware-Queues festgelegt und die Anzahl der Requests, die die Queue beinhalten kann. Wichtig und zentral: An das Tag-Set wird im Attribut `ops` die Adresse der Funktion angehängt, die für den eigentlichen Daten-Transfer zuständig ist. Das Anlegen des Tag-Sets selbst erfolgt durch Aufruf der Funktion `blk_mq_alloc_tag_set()`.

Die eigentliche Transferfunktion wertet schließlich die übergebenen Requests aus. Im Request finden sich die Transferrichtung (lesen oder schreiben, `cmd_flags`), der Sektor (`__sector`), ab dem `__data_len` Bytes zu transferieren sind, und eine Liste von `struct bio`-Elementen, die abhängig von der Transferrichtung Quelle oder Ziel der Daten im Hauptspeicher beschreiben 4.

Wir haben also für Quelle oder Ziel nicht nur eine Adressangabe, sondern mehrere. Das hängt damit zusammen, dass bei größeren Datenmengen unter Umständen im Hauptspeicher kein ausreichend großer, zusammenhängender Speicherbereich zu reservieren ist. Pragmatisch stellt der Kernel eine Liste von

Listing 1: Blockgerätetreiber für Ramdisk (Fortsetzung)

```
  dummy_disk->queue = blk_mq_init_queue( &dummy_tag_
set );
  if ( IS_ERR(dummy_disk->queue) ) {
    goto fail_blk_mq_init_queue;
  }
  blk_queue_logical_block_size( dummy_disk->queue,
512 );

  dummy_disk->major = register_blkdev( 0, "dummy_bd"
);
  dummy_disk->first_minor = 0;
  dummy_disk->fops = &bd_ops;
  snprintf(dummy_disk->disk_name, DISK_NAME_LEN,
"dummy_bd");

  set_capacity( dummy_disk, NR_SECTORS );
  printk("adding block device %s\n", dummy_
disk->disk_name );

  add_disk( dummy_disk );

  return 0;

fail_blk_mq_init_queue:
  del_gendisk( dummy_disk );
fail_alloc_disk:
  if (dummy_tag_set.tags)
    blk_mq_free_tag_set( &dummy_tag_set );
fail_alloc_tag_set:
  if ( ram_bd )
    vfree( ram_bd );
  return -EIO;
}

static void __exit dummy_bd_exit(void)
{
  struct request_queue *q = dummy_disk->queue;
  int major = dummy_disk->major;

  if (dummy_tag_set.tags)
    blk_mq_free_tag_set( &dummy_tag_set );
  del_gendisk( dummy_disk );
  put_disk( dummy_disk );
  if (q) {
    printk("blk_cleanup_queue( %p )\n", q );
    blk_cleanup_queue( q );
  }
  unregister_blkdev(major, "dummy_bd");
  if ( ram_bd )
    vfree( ram_bd );
  return;
}

module_init(dummy_bd_init);
module_exit(dummy_bd_exit);
MODULE_LICENSE("GPL");
```



4 **Eine Liste von `struct-bio`-Elementen, die abhängig von der Transferrichtung Quelle oder Ziel im Hauptspeicher beschreiben.**

Segmenten zur Verfügung, in die beziehungsweise von denen die Daten zu transferieren sind und die in Summe den notwendigen Speicher umfassen. Letztlich muss für jeden dieser Speicherbereiche separat ein Transfer initiiert werden. Bei der Programmierung dieses Auseinanderfriemelns helfen Makros wie beispielsweise `rq_for_each_segment()`, die in einer Struktur `struct bio_vec` Anfangsadresse und Länge des jeweiligen Segments liefern.

An Beispielen lernen

Zusammenhänge und Abläufe lassen sich an einem Beispieltreiber am besten nachvollziehen. Listing 1 zeigt den Code eines Blockgeräte-Treibers `dummy_bd`, der eine einfache Ramdisk definiert.

Beim Laden des Treibers wird die Funktion `dummy_bd_init()` aufgerufen, die für die Initialisierung und das Einklinken des Treibers in das Blockgeräte-Subsystem des Linux Kernels zuständig ist. Als erstes wird dazu für die Ramdisk per `vmalloc()` ausreichend Hauptspeicher abgezwackt. Daraufhin ist das Tag-Set mit der Adresse der Datenstruktur, die die Adresse der eigentlichen HW-Transferfuntkion enthält, der Anzahl der HW-Queues und deren Tiefe zu initialisieren. Die daraus resultierenden Tags und Requests werden reserviert. Anschließend folgen das Anlegen und Vorbereiten der Struktur `gendisk`. Major-Nummer, Disk-Name und Kapazität sind anzugeben. Außerdem wird das Blockgeräte-Objekt mit dem Tag-Set verknüpft. In Listing 1 ist außerdem die Implementierung der Zugriffsfunktionen auf das Blockgeräte-Objekt (`gendisk`) Open- und Release zu erkennen.

Listing 2: Makefile

```
obj-m := dummy_bd.o

KERNELDIR ?= /lib/modules/$(shell
uname -r)/build

all default: modules
install: modules_install

modules modules_install help
clean:
  $(MAKE) -C $(KERNELDIR)
M=$(shell pwd) $@
```

Der eigentliche Datentransfer, also die Abarbeitung der Requests, die der Kernel dem Treiber übergibt, ist in zwei Funktionen aufgeteilt. Die Funktion `dummy_block_request()` signalisiert durch Aufruf von `blk_mq_start_request()`, dass der Request in Bearbeitung ist. Sie überprüft, ob es sich um einen Schreib- oder einen Leseauftrag handelt, und ruft in diesem Fall die Funktion auf, die das Kopieren realisiert. Schließlich teilt sie die abgeschlossene Bearbeitung über `blk_mq_end_request()` dem Blockgeräte-Subsystem mit.

Hilfreiche Makros

In der Funktion `do_request()`, die das Kopieren der Daten übernimmt, ist der Einsatz des Makros `rq_for_each_segment()` sichtbar. Es liefert für jedes Segment die Anfangsadresse und die Größe. Das Einblenden der Adresse in den Kernelspace übernimmt die Funktion `page_address()` oder alternativ `kmap()` (in Kombination mit `kunmap()`). Die Transferrichtung ist noch zu evaluieren, woraufhin im Fall der Ramdisk per `memcpy()` die Daten verschoben werden.

Ist das Linux-System für das Kompilieren von Kernel-Code vorbereitet und sind die Kernel-Header sowie die notwendigen Werkzeuge installiert, kompilieren Sie `dummy_bd.c` mithilfe eines geeigneten Makefiles (Listing 2) durch Aufruf von `make`. Auf dem Nicht-Produktiv-System laden Sie dann mit Root-Rechten der Treiber über `sudo insmod dummy_bd.ko` in den Kernel **5**. Ein `cat /proc/devices` belegt, dass es das neue Blockgerät mit Namen `dummy_bd` gibt, das sich danach per `sudo mkfs.ext4 /dev/dummy_bd` formatieren lässt. Anschließend steht dem Einhängen und Zugreifen nichts mehr im Wege: `sudo mount /dev/dummy_bd /mnt`.

Funktionen des Blockgeräte-Subsystems

Funktion	Beschreibung
Integration ins Blockgeräte-Subsystem	
`int register_blkdev(unsigned int, const char *)`	Reservieren einer Majornumber beim Blockgeräte-Subsystem
`void unregister_blkdev(unsigned int, const char *)`	Freigabe der Majornumber
Umgang mit der zentralen `struct gendisk`	
`struct gendisk *alloc_disk(int minors)`	Reservieren einer `struct gendisk`
`void add_disk(struct gendisk *disk)`	Gerät beim Blockgerätesubsystem anmelden
`void set_capacity(struct gendisk *disk, sector_t size)`	Kapazität festlegen
`void del_gendisk(struct gendisk *gp)`	Gerät beim Blockgerätesubsystem abmelden
`void put_disk(struct gendisk *disk)`	Freigeben des per `alloc_disk()` reservierten Speichers
Tag-Sets anlegen und freigeben	
`int blk_mq_alloc_tag_set(struct blk_mq_tag_set *set)`	Anlegen eines Tag-Sets
`void blk_mq_free_tag_set(struct blk_mq_tag_set *set)`	Freigeben eines Tag-Sets
Request-Verarbeitung	
`struct request_queue *blk_mq_init_queue(struct blk_mq_tag_set *)`	Anlegen der Request Queue
`void blk_cleanup_queue(struct request_queue *)`	Freigabe der Request Queue
`void blk_mq_start_request(struct request *rq)`	Start eines Transferauftrags signalisieren
`void blk_mq_end_request(struct request *rq, blk_status_t error)`	Ende eines Transferauftrags signalisieren

Nach dem Mounten zeigt ein `ls /mnt/` das linuxtypische Verzeichnis `lost+found` an. Ein Schreiben und Lesen auf die Ramdisk ist möglich, und solange der Treiber nicht wieder entladen wird, bleiben die Daten auch über ein `umount` mit nachfolgendem erneuten `mount` erhalten.

Erst beim Entladen des Treibers über `sudo rmmod dummy_bd` wird im Rahmen der Funktion `dummy_bd_exit()` der mit `vmalloc()` reservierte Datenspeicher per `vfree()` wieder freigegeben. Außerdem wird das Blockgerät beim Blockgeräte-Subsystem abgemeldet und alle zuvor reservierten Speicherbereiche wieder freigegeben. Auf der Ramdisk lassen sich übrigens auch per `fdisk`-Kommando zwei Partitionen anlegen, da wir bei der Initialisierung die Funktion `alloc_disk()` mit dem Parameter drei aufgerufen haben.

Lesen bildet

Der Code in Listing 1 stellt einen ersten Einstieg in einen Blockgerätetreiber dar. Das ausgereifte Blockgeräte-Subsystem bietet dabei noch weitaus mehr Facetten, um einen leistungsfähigen Treiber zu erstellen, der alle denkbaren Situationen meistert (siehe Tabelle Funktionen des Blockgeräte-Subsystems). Eine übersichtliche Darstellung der Architektur findet sich übrigens in einem Vortrag von den CLT 2021. Der programmtechnische Aufbau eines Blockgerätetreibers lässt sich in einer Referenz der Linux Kernel Labs vertiefen. *(jcb)*

```
quade@ezs-cocka:~/dummy_bd$ ls
dummy_bd.c  Makefile
quade@ezs-cocka:~/dummy_bd$ make
make -C /lib/modules/5.11.0-27-generic/build M=/home/quade/dummy_bd modules
make[1]: Verzeichnis „/usr/src/linux-headers-5.11.0-27-generic" wird betreten
  CC [M]  /home/quade/dummy_bd/dummy_bd.o
  MODPOST /home/quade/dummy_bd/Module.symvers
  CC [M]  /home/quade/dummy_bd/dummy_bd.mod.o
  LD [M]  /home/quade/dummy_bd/dummy_bd.ko
make[1]: Verzeichnis „/usr/src/linux-headers-5.11.0-27-generic" wird verlassen
quade@ezs-cocka:~/dummy_bd$ sudo su
root@ezs-cocka:/home/quade/dummy_bd# insmod dummy_bd.ko
root@ezs-cocka:/home/quade/dummy_bd# cat /proc/devices | grep dummy_bd
252 dummy_bd
root@ezs-cocka:/home/quade/dummy_bd# mkfs.ext4 /dev/dummy_bd
mke2fs 1.45.5 (07-Jan-2020)
Ein Dateisystem mit 25600 (4k) Blöcken und 25600 Inodes wird erzeugt.

beim Anfordern von Speicher für die Gruppentabellen: erledigt

Inode-Tabellen werden geschrieben: erledigt
Das Journal (1024 Blöcke) wird angelegt: fertig
Die Superblöcke und die Informationen über die Dateisystemnutzung werden
geschrieben: erledigt

root@ezs-cocka:/home/quade/dummy_bd# mount /dev/dummy_bd /mnt/
root@ezs-cocka:/home/quade/dummy_bd# ls -l /mnt/
insgesamt 16
drwx------ 2 root root 16384 Aug 23 15:59 lost+found
root@ezs-cocka:/home/quade/dummy_bd# dd if=/dev/urandom of=/mnt/some-data count=
4k bs=1k
4096+0 Datensätze ein
4096+0 Datensätze aus
4194304 Bytes (4,2 MB, 4,0 MiB) kopiert, 0,0494874 s, 84,8 MB/s
root@ezs-cocka:/home/quade/dummy_bd# ls -l /mnt/
insgesamt 4112
drwx------ 2 root root   16384 Aug 23 15:59 lost+found
-rw-r--r-- 1 root root 4194304 Aug 23 15:59 some-data
root@ezs-cocka:/home/quade/dummy_bd# umount /mnt
root@ezs-cocka:/home/quade/dummy_bd# 
```

5 Der selbst erstellte Ramdisk-Treiber im Einsatz. Er kümmert sich um das Generieren, Laden, Formatieren und den Zugriff.

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/44715