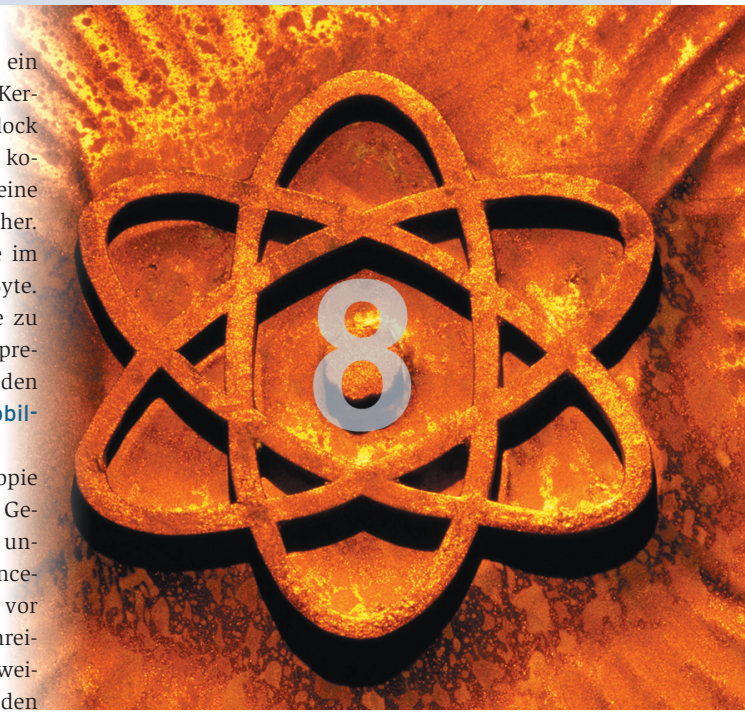
Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 8

Kern-Technik

Die Geräteklasse der Blockdevices bildet die Grundlage für Filesysteme. Diese Kern-Technik-Folge erklärt die Treiberkomponenten und zeigt die Implementierung einer RAM-Disk als Beispiel für das neue Blockgeräte-Interface in Kernel 2.6. Eva-Katharina Kunst, Jürgen Quade



Der Name klingt zwar sperrig, doch Blockgeräte haben mit großen Kisten nichts zu tun. Festplatten, DVDs, CD-ROMs, USB-Sticks oder Flashcards verarbeiten Daten blockweise und verlangen deshalb nach einem Blockgeräte-Interface im Kernel. Meist dienen sie dazu, Daten dauerhaft zu speichern, also als Hintergrundspeicher, dessen Verkörperung eben eine Festplatte ist. Programme können mit Funktionen wie »fseek()« oder »lseek()« an beliebige Stellen auf dem Blockgerät springen und dort lesen oder schreiben (Random Access).

Am Ende anfangen

Dabei lässt sich das letzte Byte also durchaus vor dem ersten lesen. Vergleicht man den Transport von Daten mit fließendem Wasser, handelt es sich bei einem Blockgerät nicht um einen Datenstrom, sondern eher um ein stehendes Gewässer. Wer hier Wasser schöpfen will, kann dazu nur einen Eimer benutzen – selbst dann, wenn er lediglich einen einzigen Wassertropfen benötigt. Beim Kernel 2.6 fasst ein solcher Eimer genau 512 Bytes. Wer von einem Blockgerät ein einzelnes Byte lesen möchte, muss also stets 512 Bytes entnehmen. Die restlichen 511 Bytes lässt er dann einfach liegen.

Da von einem Blockgerät ohnehin keine einzelnen Bytes gelesen werden können, verwaltet der Kernel den Hintergrundspeicher über Blocknummern, unabhängig davon, dass auf Hardware-Ebene Köpfe, Zylinder und Sektoren dahinter stecken. Für die ersten 512 Bytes eines Hintergrundspeichers ist Blocknummer 0 maßgeblich, Bytes 512 bis 1023 entsprechen Blocknummer 1 und so weiter.

Ist von einem Blockgerät ein Byte zu lesen, berechnet der Kernel zunächst, in welchem Block sich das Byte befindet, und kopiert diesen Block über seine Nummer in den Hauptspeicher. Dann liest er aus der Kopie im Speicher das gewünschte Byte. Lautet der Auftrag, ein Byte zu schreiben, liest er den entsprechende Block ebenfalls in den Hauptspeicher ein (siehe Abbildung 1).

Nach der Änderung der Kopie schreibt er den Block auf das Gerät zurück. Allerdings nicht unbedingt sofort: Aus Performancegründen wartet das System vor dem physischen Zurückschreiben erst einmal ab, ob noch weitere Schreibaufträge für den Block eintreffen.

Namen statt Zahlen

Damit der Anwender Daten auf dem Blockgerät ablegen und wiederfinden kann, verwaltet der Kernel die Blöcke eines solchen Geräts über ein Dateisystem, beispielsweise Ext 3 (siehe auch Titelthema-Artikel). Durch selbst vergebene Namen für Verzeichnisse und Dateien ist der Zugriff für den Anwender einfacher, als wenn er Blocknummern verwenden müsste.

Auch dem Treiberprogrammierer bringt die Abstraktion Vorteile. Er muss sich nicht darum kümmern, was eine Anwendung auf der Ebene des Dateisystems macht. Sein Treiber bekommt stattdessen vom Betriebssystem so simple Aufträge wie „Kopiere den Inhalt von Block xyz in die Mempage abc“ oder „Schreibe den Inhalt von Mempage abc in Block xyz“.

Um vernünftige Aufträge zu erteilen, braucht der Kernel einige Informationen über den Hintergrundspeicher, etwa dessen Größe angegeben als Anzahl der Blöcke. Außerdem muss er die Major- und Minor-Nummer kennen, um mit ihrer Hilfe über die Gerätedatei die Zuordnung zwischen Applikation respektive Dateisystem und Treiber herzustellen (siehe Kasten „Gerätenummern“).

Datenstrukturen

Alle Informationen stecken in der Datenstruktur »struct gendisk« (definiert im Header »linux/genhd.h«). Diese Struktur ist für den Blockgerätetreiber von zentraler Bedeutung, denn sie spezifiziert sowohl das Gerät – die Disk – als auch

die Gerätezugriffe über den Treiber. Neben der Sektorgröße und der Major- und Minor-Nummer gehören auch der Gerätename und der auf dem Gerät vorhandene Speicherplatz zu den Elementen, die der Treiberentwickler angibt.

Treiberkomponenten

Die Routine »struct gendisk« ist nur eine von drei elementaren Komponenten eines Blockgerätetreibers. Die zweite ist eine Datenstruktur, die der Kommunikation zwischen Treiber und Blockgeräte-Subsystem dient: die Request-Queue (siehe Abbildung 2). In einer solchen Queue sammelt der Kernel alle Aufträge an den Treiber samt dem Zeiger auf die Daten. Mehr noch, der Kernel sortiert in einer so genannten Make-Request-Funktion die Aufträge (Requests) der »request_queue« nach der Sektornummer. So arbeitet ein Treiber physisch nahe zusammenliegende Sektoren nacheinander ab und spart damit Zeit.

Die dritte Komponente ist schließlich die Funktion, die die Befehle „Kopiere den Inhalt von Block xyz in die Mempage abc“ oder „Schreibe den Inhalt von Mempage abc in Block xyz“ ausführt, also den eigentlichen Datentransfer abwickelt. Sie heißt Request-Funktion und ist vom Typ »request_fn_proc«. Im einfachsten Fall arbeitet Request-Funktion die Aufträge der Request-Queue nacheinander ab. Die verschiedenen Auftragstypen sind in der Headerdatei »linux/blkdev.h« zu finden.

Die Requests selbst sind einen genaueren Blick wert: Ein einziger Auftrag überträgt zugleich mehrere Blöcke in den Hauptspeicher. Dann wird es jedoch komplizierter, denn für mehrere Blöcke im Hintergrundspeicher müssen auch mehrere Blöcke im Hauptspeicher vorhanden sein. Dort liegen sie aber, im Unterschied zum Hintergrundspeicher, nur selten sequenziell hintereinander.

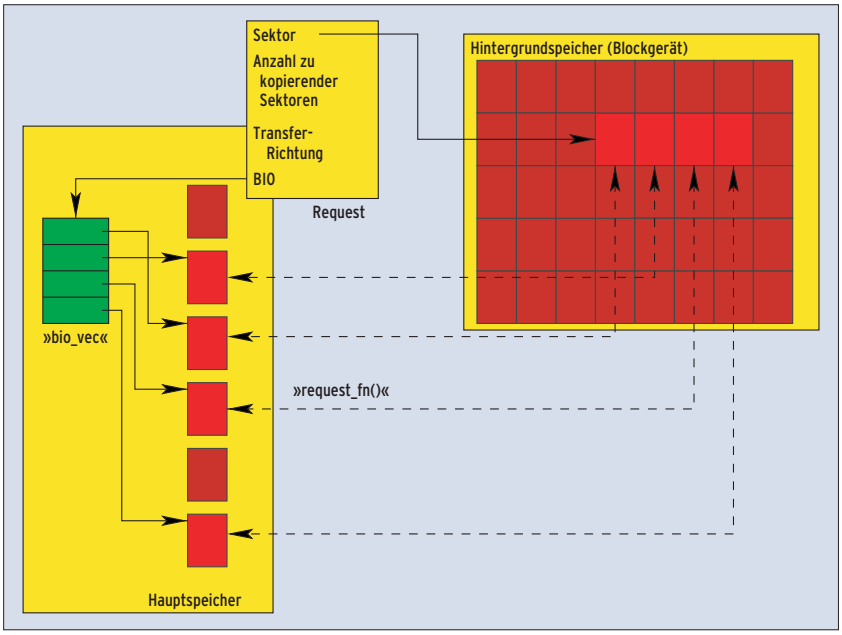


Abbildung 1: Der Kernel verwendet für den Zugriff auf Daten eines Blockgeräts Kopien der Blöcke im Hauptspeicher. Dafür stellt ein Treiber die Funktion »request_fn()« bereit.

Alles BIO

Zur Spezifikation der Blöcke im Hauptspeicher werden daher mit jedem Request eine oder mehrere so genannte BIO-Strukturen (Block I/O) verknüpft, siehe Abbildungen 2 und 3. Jede BIO-Struktur besitzt ein Feld von Zeigern auf Speicherseiten namens »bio_vec«. Liegen die Speicherseiten aber im Userspace oder im High-Memory-Bereich, sind sie nicht direkt adressierbar. Eine Adressierung über die Page ermöglicht in diesen Fällen aber einen Transfer per DMA.

Da bei einem virtuellen Gerät die Optimierung einzelner Requests unnötig ist, kann man die systemeigene Make-Request-Funktion »make_request()« durch eine eigene Funktion ersetzen. Sie wickelt den Datentransfer direkt ab und macht eine (normale) Request-Funktion unnötig. Damit entfällt auch das Iterieren über die einzelnen Requests: Die ersetzte Make-Request-Funktion bekommt die Aufträge – sobald sie anfallen – in einem BIO-Block übergeben.

Ran an die Tastatur

So viel zur Theorie, jetzt zur Praxis. Wenn der Kern eines Blockgerätetreibers nur aus dem Transfer von Blöcken be-

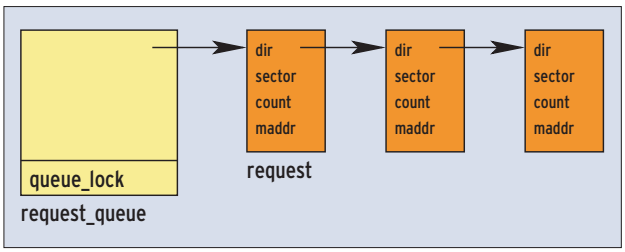


Abbildung 2: Request-Datenstruktur: Eine Request-Queue enthält die unterschiedlichen Aufträge in Form einer verketteten Liste. Der Zugriff auf einzelne Listenelemente erfolgt über das Makro »elv_next_req()«.

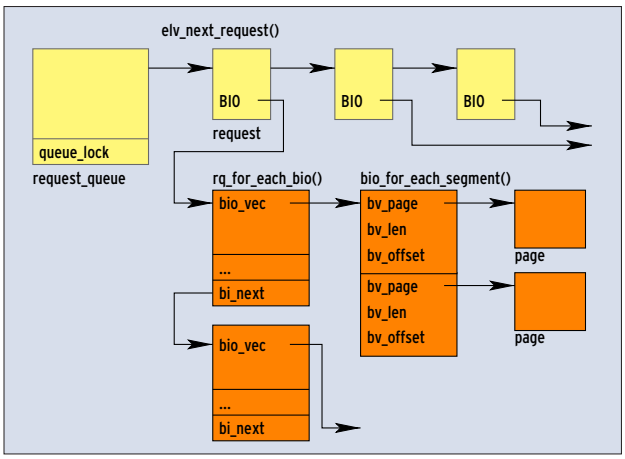


Abbildung 3: Die BIO-Datenstruktur enthält Felder von Zeigern »bio_vec«, die letztlich auf Seiten im Hauptspeicher verweisen.

steht, kann es nicht so schwer sein, einen Treiber für eine virtuelle Festplatte (RAM-Disk) zu schreiben.

Der Speicher der virtuellen Disk soll gleich beim Laden des Treibers reserviert werden. Die Funktion »kmalloc()« eignet sich dafür nicht, weil sie nur maximal 64 KByte zusammenhängenden Speicher reserviert. Stattdessen kommt »vmalloc()« zum Einsatz, das hier 8 MByte alloziert, was für die RAM-Disk genügt. Die spätere Freigabe des Speichers erfolgt mit »vfree()«.

Zugriffsrechte vereinfacht

Die Funktion »alloc_disk()« (Listing 1, Zeile 52) reserviert Speicher für die »struct gendisk«. In diese trägt der Treiber die Kapazität der virtuellen Festplatte und die Tabelle mit den Funktionszeigern ein. Da Zugriffsrechte in diesem Beispiel keine besondere Rolle spielen, reicht es bereits aus, in der Tabelle lediglich das Feld ».owner« zu belegen. Dieses Feld muss der Treiber allerdings in jedem Fall ausfüllen, damit der Kernel das Modul erst dann zum Entladen freigibt, wenn keine Instanz mehr darauf zugreift.

Bevor der Aufruf von »add_disk()« (Zeile 63) das Objekt dem Blockgeräte-Subsystem übergibt, muss die Request-Queue erzeugt werden. Der Treiber für ein virtuelles Gerät reserviert den Speicher für die Request-Queue durch Aufruf von »blk_alloc_queue()« (Zeile 50) und überschreibt die vorgegebene Make-Request-Funktion durch die eigene Variante »bdMakeRequest()«. Dazu ruft der Treiber »blk_queue_make_request()« auf (Zeile 51). Eine gesonderte Request-Funktion ist in diesem Fall nicht nötig. Der noch in Kernel 2.4 notwendige Aufruf der Funktion »register_blkdev()« ist in Kernel 2.6 optional (siehe hierzu Kasten „Portierungshilfe“).

Die Funktion »bdMakeRequest()« bekommt die Request-Queue »q« und den Zeiger auf den BIO-Block des aktuellen Requests »bio« übergeben. Den Parameter »q« verwendet die Funktion nicht, denn der Treiber verarbeitet Anfragen sofort, ohne sie in eine Schlange einzureihen. Bevor der Treiber »bio« auswertet, muss er sicherstellen, dass er auf den Datenbereich des BIO-Blocks zugreifen darf – mit der Funktion »blk_queue_bounce()« (Zeile 25) kein Problem. Um jetzt die Daten zwischen virtueller Festplatte und dem BIO-Block per »memcpy()« zu transferieren, muss der Treiber noch die Quell- und Zieladressen sowie die Anzahl der zu kopierenden Bytes bestimmen.

Adressen berechnen

Über die Startadresse der RAM-Disk und den durch die Sektornummer gegebenen Offset lässt sich die Adresse »maddr« auf der virtuellen Festplatte berechnen. Um an die entsprechenden Adressen im Hauptspeicher zu kommen, durchsucht der Treiber die Tabelle »bio_vec«, in der die Adressangaben des BIO-Blocks stehen. Hierbei hilft das Makro »bio_for_each_segment()«.

Die Makros »bio_data()« und »bio_data_dir()« lesen schließlich die Adresse des zugehörigen Datenbereichs und die

Listing 1: Treiber einer virtuelle Festplatte

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/blkdev.h>
06
07 #define BD_MAJOR 242
08 #define SIZE_IN_KBYTES (8*1024)
09
10 MODULE_LICENSE("GPL");
11
12 static struct gendisk *disk;
13 static struct request_queue *bdqueue;
14 static char *DiscSpace;
15 static struct block_device_operations bdops = {
16     .owner          = THIS_MODULE,
17 };
18
19 static int bdMakeRequest( request_queue_t *q,
   struct bio *bio )
20 {
21    char *kaddr, *maddr;
22    struct bio_vec *bvec;
23    int segnr, BytesTransferred=0;
24
25    blk_queue_bounce( q, &bio );
26    bio_for_each_segment( bvec, bio, segnr ) {
27       kaddr = bio_data(bio);
28       maddr = DiscSpace + (512 * bio-
   >bi_sector);
29       BytesTransferred += bio->bi_size;
30       if( bio_data_dir( bio )==READ ||
   bio_data_dir( bio )==READA ) {
31          memcpy( kaddr, maddr, bio->bi_size );
32       } else {
33          memcpy( maddr, kaddr, bio->bi_size );
34       }
35    }
36    bio_endio( bio, BytesTransferred, 0 );
37    return 0;
38 }
39
40 static int __init ModInit(void)
41 {
42    if( register_blkdev(BD_MAJOR, "bdsample" ))
   {
43       printk("blockdevice: Majornummer %d not
   free.", BD_MAJOR);
44       return -EIO;
45    }
46    if(
   !(DiscSpace=vmalloc(SIZE_IN_KBYTES*1024)) ) {
47       printk("vmalloc failed ...\n");
48       goto out_no_mem;
49    }
50    bdqueue = blk_alloc_queue( GFP_KERNEL );
51    blk_queue_make_request( bdqueue,
   bdMakeRequest );
52    disk = alloc_disk(1);
53    if( !disk ) {
54       printk("alloc_disk failed ...\n");
55       goto out;
56    }
57    disk->major = BD_MAJOR;
58    disk->first_minor = 0;
59    disk->fops = &bdops;
60    disk->queue = bdqueue;
61    sprintf(disk->disk_name, "bd0");
62    set_capacity( disk, (SIZE_IN_KBYTES*1024)>>9
   ); // in 512 Byte Bloecke
63    add_disk( disk );
64    return 0;
65 out:
66    vfree( DiscSpace );
67 out_no_mem:
68    unregister_blkdev(BD_MAJOR, "bdsample" );
69    return -EIO;
70 }
71
72 static void __exit ModExit(void)
73 {
74    unregister_blkdev(BD_MAJOR, "bdsample" );
75    del_gendisk(disk);
76    put_disk( disk );
77    blk_cleanup_queue( bdqueue );
78    vfree( DiscSpace );
79 }
80
81 module_init( ModInit );
82 module_exit( ModExit );
```

Transferrichtung aus. Sie beziehen sich immer auf das aktuelle Segment, also auf den Feldeintrag in »biovec«. Dass der Treiber den Auftrag beendet hat, signalisiert er mit »bio_endio()«. Diese Funktion hat als Parameter einen Zeiger auf den bearbeiteten BIO-Block, die Anzahl der transferierten Bytes und einen möglichen Fehlercode. »BdMakeRequest()« liefert »0« zurück, als Zeichen dafür, dass der Transfer erfolgreich war.

Nach getaner Arbeit muss der Blockgerätetreiber aufräumen. Mit »put_disk()« meldet er die Platte, mit »unregister_blkdev()« sich selbst beim Kernel ab. Dabei gibt er die zu Beginn reservierten Ressourcen für die Platte mit »del_gendisk()« und für die Queue durch »blk_cleanup_queue()« frei. Vor der Freigabe der Queue muss der Treiber »struct gendisk« freigeben.

Mit Hilfe eines passenden Makefile (siehe Listing 2) lässt sich der RAM-Disk-Treiber einfach kompilieren. »insmod« lädt das Modul in den Kernel:

```
insmod virtdisc.ko
```

Um die virtuelle Disk zu testen, ist zunächst eine Gerätedatei anzulegen. Vor dem Mounten muss der Root-User auf der virtuellen Platte ein Filesystem erstellen:

```
mknod virtdiscfile b 242 0
mke2fs virtdiscfile
mount virtdiscfile /mnt
```

Wer versucht das Blockgerät zu partitionieren, wird wenig Erfolg haben. Der Beispieltreiber hat sich nämlich beim Kern nur für ein einziges Gerät registriert. Der Parameter der dafür verwendeten Funktion »alloc_disk()« gibt die Anzahl der unterstützten Geräte an. Er muss also, soll sich das Gerät partitionieren lassen, einen Wert größer als 1 erhalten. Möchten Programme wie »fdisk« die Geometrie der Platte abfragen, ist im Treiber das IO-Control »HDIO_GETGEO()« zu implementieren (siehe [3]).

Listing 2: Makefile für die virtuelle Harddisk

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m   := virtdisc.o
03
04 else
05 KDIR    := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09         $(MAKE) -C $(KDIR)      SUBDIRS=$(PWD) modules
10
11 clean:
12         rm -f *.ko *.o *.mod.c
13 endif
```

Reale Geräte

Ein Blockgerätetreiber für reale Geräte unterscheidet sich von dem vorgestellten Code in zweierlei Hinsicht:

- Ein reales Gerät verwendet die Funktion »blk_init_queue()« , um einer Request-Queue die Adresse der Request-Funktion zu übergeben und sich die Adresse der zugehörigen Request-Queue geben zu lassen. Es gibt kein »blk_alloc_queue()« und kein »blk_queue_make_request()«. Die Funktion erwartet als zweiten Parameter die Adresse eines Spinlocks, der parallele Zugriffe auf die Request-Queue absichert:

```
if( (bdqueue = blk_init_queue(
   &bdRequest, &bdlock))==NULL )
      goto out;
```

- Ein echter Blockgerätetreiber realisiert eine Request-Funktion (siehe Abbildung 4).

Die Request-Funktion entnimmt mit Hilfe des Makros »elv_next_request()« den jeweils nächsten Auftrag aus der Request-Liste:

```
while((req = elv_next_request(q))
   != NULL ) {
```

Das Makro erhält als Parameter die Request-Queue und gibt einen Zeiger auf den nächsten Auftrag »req« zurück (vom Typ »struct request *«, siehe Abbildung 2). Zunächst muss der Treiber aber feststellen, ob es sich um einen normalen Schreib- oder Leseauftrag handelt:

Gerätenummern

Mit Kernel 2.6 hat Linus Torvalds das Ende der klassischen Major- und Minor-Nummern eingeläutet. Anders als geplant sind die neuen Gerätenummern aber nicht 64, sondern nur 32 Bit breit. Abbildung 5 zeigt, wie die alten Major- und Minor-Nummern auf die neuen Gerätenummern abgebildet werden. Das im Artikel vorgestellte Treiberinterface arbeitet weiterhin nach dem alten Schema.

Wer die neuen Gerätenummern nutzen möchte, muss auf zusätzliche Funktionen zurückgreifen. Dazu meldet der Treiber sein Interesse an den benötigten Gerätenummern durch den Aufruf der Funktion »blk_register_region()« an:

```
void blk_register_region(
   dev_t dev,
   unsigned long range,
   struct module *module,
   struct kobject *(*probe)(dev_t,int
*,void *),
   int (*lock)(dev_t,void *),
   void *data);
```

Allerdings sind damit die Gerätenummern nicht sofort exklusiv reserviert. Vielmehr informiert der Kernel den Treiber, sobald er auf eine Disk im entsprechenden Bereich zugreifen möchte, durch Aufruf von »probe()«. Diese Funktion sucht die passende »struct gendisk« und gibt einen Zeiger auf das in der Struktur eingebettete »kobject« zurück. Hat der Programmierer außerdem die Adresse einer Funktion »lock()« übergeben, ruft der Kernel diese zuvor auf.

Der Datentyp »dev_t« repräsentiert die neue Gerätenummer. Das Makro »MK_DEV(major, minor)« generiert aus der Major-Nummer »major« und der Minor-Nummer »minor« die Gerätenummer. Der Parameter »range« von »blk_register_region()« gibt die Anzahl der zu reservierenden Gerätenummern an.

Wird der Treiber deinitialisiert, nimmt er durch Aufruf von »blk_unregister_region(dev_t dev, unsigned long range)« die Reservierung wieder zurück (siehe auch [4]).

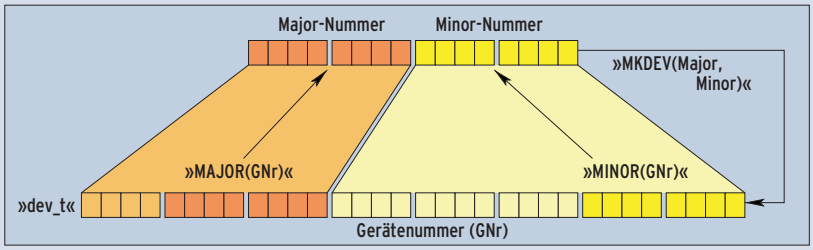


Abbildung 5: Eine eindeutige Vorschrift bildet die alten Major- und Minor-Nummern auf die neuen Gerätenummern ab. »MKDEV()« erzeugt aus Major/Minor eine Gerätenummer, die Makros »MAJOR()« und »MINOR()« sind für die Gegenrichtung zuständig.

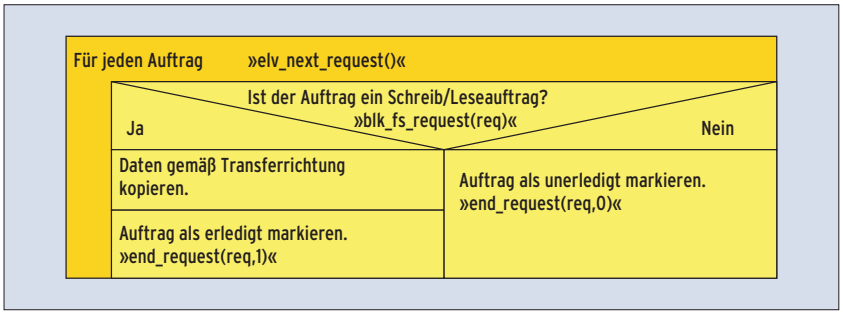


Abbildung 4: Das Struktogramm zeigt die Arbeitsweise der Transferfunktion eines Blockgerätetreibers. Die Routine »elv_next_request()« bildet eine Schleife für alle Aufträge an den Treiber.

```
if( !blk_fs_request(req) ) {
   // kein Schreib-/Leseauftrag
   end_request( req, 0 );
      continue;
}
```

Ist dies nicht der Fall, holt der Treiber mit »continue« den nächsten Auftrag aus der Liste. Handelt es sich um einen Schreib- oder Leseauftrag, arbeitet er ihn ab. Das Makro »rq_for_each_bio()« liefert dazu als Teil einer Schleife den jeweiligen BIO-Block zurück, den der Treiber dann wie beschrieben auswertet:

```
rq_for_each_bio(bio, req ) {
```

Hat er einen Auftrag abgearbeitet, ruft er »end_request()« auf. Der erste Parameter ist dabei der Auftrag selbst. Der zweite Parameter spezifiziert mit einem Wert ungleich null, dass der Auftrag bearbeitet wurde (siehe Abbildung 4).

Rückblick und Vorschau

Das Blockgerätemodell bietet eine einheitliche Programmierschnittstelle für reale Hardware zur Datenspeicherung – von der großen Festplatte bis zum USB-Stick. Um USB-Geräte geht es in der nächsten Folge der Kern-Technik, denn die haben mittlerweile RS-232 als Standardschnittstelle abgelöst. *(ofr)* ■

Infos:

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", Folge 1: Linux-Magazin 8/03, S. 88

[2] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur": Linux-Magazin 1/04, S. 28

[3] Jonathan Corbet, „Driver Porting: A simple block driver": [http://lwn.net/Articles/58719/]

[4] Jonathan Corbet, „Driver Porting: the gendisk interface": [http://lwn.net/Articles/25711/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Portierungshilfe

Im Kernel 2.6 repräsentiert nicht mehr die Tabelle der Treiber-Einsprungspunkte »struct block_device_operations« das Blockgerät, sondern die Struktur »struct gendisk«. Die Tabelle mit den Treiber-Einsprungspunkten trägt man nun in die »struct gendisk« ein, die darüber hinaus die Kenndaten des Geräts (der Disk) aufnimmt. Die dafür vorgesehenen globalen Variablen aus Kernel 2.4 existieren nicht mehr.

Die Übergabe des Objekts an das Blockgeräte-Subsystem hat sich ebenfalls geändert. Sie findet nicht mehr in »register_blkdev()«, sondern in der Funktion »add_disk()« statt. Außerdem braucht Linux 2.6 die Funktion »register_blkdev()« nicht mehr. Nur wer seinem Treiber dynamisch eine Minor-Nummer zuteilen lassen und ihn im Verzeichnis »/proc/devices« listen möchte, verwendet die Funktion weiterhin.

Lokale Queues

Auch die interne Pufferung hat sich verändert. Statt einer globalen Request-Queue bringt jedes Blockgerät seine eigene Warteschlange und folglich auch seinen eigenen Lock mit. Den Lock »io_request_lock« gibt es nicht mehr.

Der Zugriff auf die Requests erfolgt nicht mehr über das Makro »CURRENT«, sondern über »elv_next_request()«. Die aus Kernel 2.4 bekannten Buffer-Heads ersetzt Version 2.6 durch die BIO-Blöcke mit ihren eigenen Zugriffsfunktionen.