Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 12

Kern-Technik

Linux ist stabil, weil sich die meisten Programmierer sehr um die Qualität ihres Codes bemühen. Statt hässlicher Quick-and-dirty-Hacks dominiert liebevoll ausgearbeitete Software die zentralen Teile. Im Kernel ist dies besonders wichtig, weil es folgenreiche Kernel-Bugs vermeidet. Eva-Katharina Kunst, Jürgen Quade

Software im Allgemeinen und speziell Systemsoftware muss sicher, effizient, performant, skalierbar, wartbar und anwendbar sein (siehe Tabelle 1). Einige Grundregeln für gutes Design helfen jedem Kernelprogrammierer dabei, in diesem anspruchsvollen Bereich Qualitätssoftware zu entwickeln.

Die erste Regel fordert Vollständigkeit. Ein Treiber muss alle Funktionen seiner Hardware nutzen und das Applikationsinterface komplett unterstützen. Ein Treiber, der nur »read()« und »write()«, aber kein »poll()« oder »select()« kennt, ist nur eingeschränkt tauglich. Gleiches gilt für die Zugriffsmodi. Bis zum 2.4er Kernel gab es nur blockierende und nicht blockierende, Kernel 2.6 kennt zusätzlich asynchrone Zugriffe.

Vollständig sollte auch die Integration neuer Komponenten ausfallen. Im Kernel bedeutet das: Ein Modul muss das neue Gerätemodell unterstützen sowie das Proc-Filesystem, die Hotplug-Funktionen und das Device-Filesystem. Dabei sammelt sich eine Menge Code an, der vollständig dokumentiert sein will.

Ein gutes Stück Code zeichnet sich durch einfache Handhabung aus. Falls es sich parametrisieren lässt, sollte ein Modul die Parameter nach Möglichkeit automatisch bestimmen oder zumindest mit sinnvollen Defaultwerten besetzen. Solche Komfortfunktionen tragen wesentlich zur Akzeptanz einer Softwarekomponente bei.

Beim Entwurf der Interfaces muss der Entwickler dafür sorgen, dass die Benutzer das Interface intuitiv anwenden können. Gut ist es, wenn sie einen Treiber mit »cat« und »echo« testen können. Schlechter wäre, falls sie dafür immer eine spezielle Software benötigten oder viele IO-Controls (per »ioctl()« in einem C-Programm) setzen müssten.

Einfache Handhabung

Um IO-Controls zu vermeiden, ist es oft nötig, ein physikalisches Gerät in mehrere logische Geräte einzuteilen oder viele Attributdateien im Sys- oder Proc-Filesystem zu erstellen. Während der Datenaustausch über das normale Device läuft, sind Konfiguration und Statusabfrage mit Ascii-Kommandos über die Attributdateien möglich. Wer dennoch IO-Controls implementiert, sollte die zugehörigen Kommandos normgerecht kodieren. Der Standard ist in dem Kasten „Kodierung von IO-Controls“ nachzulesen.

Im Kernel-Quellcode ist meist jede Komponente in einer eigenen Quelldatei implementiert. Sie kann auf mehrere tausend Zeilen anwachsen. Den Code auf mehrere Files verteilen ist sinnvoll, wenn es zu einer besseren Skalierbarkeit führt. Der Treiber für eine Hardware beispielsweise, die es in der Ausprägung PCI und USB gibt, lässt sich gut in drei Quelldateien und damit auch drei Modulen realisieren: Ein Modul für die Hardware-Anbindung per PCI, eins für USB und ein auf beide aufsetzendes Modul, das die Funktionalität und die Schnittstelle zur Applikation realisiert.

Wer die Hardware mit USB-Schnittstelle verwendet, belegt keinen Speicher für die PCI-Anbindung. Umgekehrt verschwendet der Besitzer einer PCI-Hardware keine Ressourcen für den USB-Code (siehe Abbildung 1).

Bevor ein angehender Kernelentwickler seine Ideen und Entwürfe in Code umsetzt, ist die Textdatei »Documentation/

Tabelle 1: Qualitätssoftware

Ziel	Programmiertechnik
Sicherheit	Sicherheitsorientierte Programmierung • Fail-Safe-Verhalten • Hilfsfunktionen nutzen
Intuitive Nutzung	Einteilung in logische Geräte • API vollständig unterstützen • Umfassende Systemintegration • Automatische Hardware-Erkennung • IO-Controls sparsam verwenden • Konfiguration optional
Zeitverhalten	Kritische Abschnitte kurz
Skalierbarkeit	Modularisierung
Sonstiges	Dokumentation • Programmierstil

CodingStyle« Pflichtlektüre. Um Quelltexte besser lesen und besser pflegen zu können, geben Linus Torvalds und seine Helfer eine Reihe von Kodierungsrichtlinien vor. Dabei halten sie sich stark an den K&R-Stil (Kernighan und Ritchie, die Entwickler der Programmiersprache C). In diesem Stil wurde bereits das Ur-Unix programmiert. Er zeichnet sich durch hohe Codedichte aus.

Im Kernel setzen die Entwickler aus Effizienz- und Performance-Gründen gelegentlich Konstrukte ein, die einem Anwendungsprogrammierer die Haare zu Berge stehen lassen. Ein richtig verwendetes »goto« spart Code, Laufzeit und erhöht sogar die Verständlichkeit. Auf Rekursion ist dagegen zu verzichten – dem Kernel steht nur ein eingeschränkter Stack zur Verfügung. Das erklärt auch, warum Kernelfunktionen nur wenig lokale Daten reservieren dürfen.

Makros wie »likely()« und »unlikely()« helfen dem Compiler bei der Optimierung. Sie umrahmen die If-Bedingung:

```
if (likely(len>1)) {
  /* tritt wahrscheinlich ein */
} else {
  /* eher unwahrscheinlich */
}
```

So weiß der Compiler, dass die Bedingung fast immer erfüllt ist, und optimiert den Code dafür. Der Kasten „Do-While-0-Makros“ erklärt ein weiteres abenteuerliches Konstrukt.

Besonders im Kernel müssen Programmierer sicherheitsbewusst arbeiten. Gegen Puffer-Überläufe hilft es, die Funktion »snprintf()« anstelle von »sprintf()« zu verwenden und »strlcpy()« statt »strcpy()«. Besonders sorgfältig ist der Code auf kritische Abschnitte zu überprüfen [1]; jeder Zugriff auf eine globale Variable ist ein Kandidat dafür. Auch die Synchronisation zwischen Codeteilen gilt es zu prüfen und gegebenenfalls zu schützen. Linux gibt dem Entwickler dazu Spinlocks, Semaphore und Completion-Objekte an die Hand. Wichtig ist auch, dass ein Modul den Wertebereich der ihm übergebenen Parameter kontrolliert.

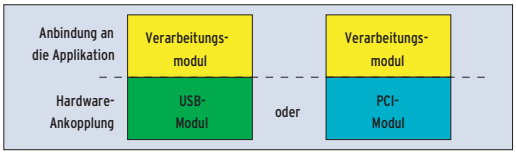

Abbildung 1: Gibt es ein Gerät in mehreren Varianten (hier mit USB- und PCI-Schnittstelle), dann ist ein modularer Gerätetreiber zu bevorzugen. Das Verarbeitungsmodul ist immer erforderlich, aber USB- und PCI-Modul sind austauschbar.

Sicherheit im Treiber

Fehler vermeidet, wer getestete Codeteile verwendet. Bietet der Kernel eine Standardfunktion für eine Aufgabe, sollte ein Treiber sie auch nutzen. Manchmal entstehen dabei aber Abhängigkeiten, die den Vorteil aufheben. Verwendet ein Treiber beispielsweise eine im ACPI-Subsystem definierte Hilfsfunktion, dann lässt er sich nur übersetzen, wenn auch ACPI konfiguriert ist.

Ein Modul muss sich auch sicherheitsbewusst verhalten. Dazu ist bei einem Treiber beispielsweise die Funktion »driver_close()« zu implementieren. Sobald sich die den Treiber nutzende Applikation beendet (etwa durch einen Fehler), ruft der Kernel »driver_close()« auf. Diese Funktion sollte Fail-Safe-Verhalten implementieren, also die vom Treiber bediente Hardware in einen sicheren Zustand bringen. Ähnlich kann ein Watchdog zu mehr Sicherheit führen: Er schaltet sich ein, sobald wichtige Aktionen nicht rechtzeitig erfolgen.

Eigene Daten halten

In Kernelcode zu speichernde Daten lassen sich in vier Kategorien einteilen:

- Treiberspezifische Daten, zum Beispiel der Treibername.
- Daten zur Hardware, beispielsweise Interruptnummer oder IO-Bereiche.
- Daten, die zu einem logischen Gerät gehören, etwa die Anzahl aktuell auf das Gerät zugreifender Instanzen.
- Daten, die zu einer Treiberinstanz gehören. Eine Instanz entsteht, wenn eine Applikation »open()« ausführt. Der Kernel entfernt die Instanz, wenn die Anwendung »close()« aufruft.

Zu jeder Kategorie passen andere Datenstrukturen. Nutzt ein Modul geeignete Strukturen, dann muss es sich nicht selbst um die Verwaltung der Datenobjekte kümmern. Viele Datenstrukturen des Kernels enthalten bereits Speicherplatz für einen Zeiger, den eine selbst erstellte Komponente nutzen darf. Im »struct file«-Objekt der Treiberinstanz heißt dieser Zeiger beispielsweise »void *private_data«. Ruft ein Rechenprozess »open()« auf, reserviert die »driver_open()«-Funktion des Moduls Speicher

Kodierung von IO-Controls

Wer IO-Controls verwendet, muss die einzelnen Kommandos, deren Parameter und ihr Verhalten (besonders im Fehlerfall) spezifizieren und dokumentieren. Für die Kodierung der IO-Control-Kommandos haben die Linux-Entwickler einen Standard festgelegt und Makros definiert, siehe »asm/ioctl.h«. Zwar ist kein Entwickler gezwungen sich daran zu halten, zu empfehlen ist es aber: Der Standard sorgt für eindeutige, unterscheidbare Kommandos innerhalb des Kernels und vereinfacht das Debugging.

Ein IO-Control-Kommando besteht aus 32 Bit (4 Byte). Dieses 32-Bit-Wort ist in vier Felder aufgeteilt (siehe Abbildung 2). Die Nummer kennzeichnet das Kommando. Mit 8 Bit lassen sich innerhalb eines Treibers 256 unterschiedliche Kommandos realisieren. Für den Typ stehen ebenfalls 8 Bit zur Verfügung. Er soll helfen IO-Controls systemweit eindeutig zu halten. Die leider veraltete Datei »Documentation/ioctl-numbers.txt« listet die bereits vergebenen Typen. Üblich ist es, den Anfangsbuchstaben des Modulnamens als Basis für diese Zahl zu verwenden.

Im IO-Control-Kommando ist auch die Länge der maximal zu transferierenden Daten kodiert. Hierzu stehen 14 Bit zur Verfügung. Null bedeutet, dass mehr als 16384 Bytes (2^{14}) übertragen werden. Für die Transferrichtung (2 Bit) gibt es vier Möglichkeiten:

- Keine Daten übertragen, der Wert ist im Makro »_IOC_NONE« abgelegt
- Daten nur lesen, Makro »_IOC_READ«
- Daten nur schreiben, Makro »_IOC_WRITE«
- Daten lesen und schreiben: »_IOC_READ | _IOC_WRITE«

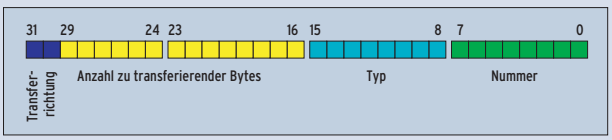

Abbildung 2: Standardkonforme IO-Control-Kommandos enthalten vier Informationen: Transferrichtung, Anzahl zu übertragender Bytes, Typ und Nummer. Mit dem Makro »_IOC(dir, type, nr, size)« vereinfacht Linux die Handhabung der IO-Controls.

für eigene Daten dieser Instanz. Die Adresse hängt es in das File-Objekt der Treiberinstanz ein (Listing 1).
Bei den darauf folgenden Zugriffen (zum Beispiel »read()«) kann das Modul per »instanz->private_data« auf seine Daten zugreifen. Der Kernel übergibt jeder Funktion einen Zeiger auf das Instanz-Objekt. Das Modul muss lediglich dafür sorgen, dass es den mit »kmalloc« reservierten Speicher später wieder freigibt.

Sparsamkeit

Wer sich um Skalierbarkeit sorgt, sollte optionale Codeteile mit »#define« ausmaskieren, beispielsweise Code für das Proc-Filesystem, das PCI-Subsystem, das Device-Filesystem oder das PNP-Subsystem (Plug&Play). Die Funktionsaufrufe dieser Kernelkomponenten sind so programmiert, dass sich Code auch kompilieren lässt, wenn die zugehörige Komponente fehlt (in der Kernelkonfiguration ausgelassen wurde). Wie Abbildung 3 verdeutlicht, spart das Ausmaskieren Code und damit Laufzeit.
Kernelcode bestimmt das Zeitverhalten des Betriebssystems. Unnötige Schleifen und lange kritische Abschnitte sind zu vermeiden. Interrupt-Service-Routinen müssen kurz sein, längere Berechnungen gehören in ein eigenes Tasklet [2]. Besseres Zeitverhalten erreichen auch Programmierer, die einen kurzen kritischen Abschnitt per Spinlock schützen statt mit einem Semaphor (Mutex) [1]. Zudem sollten sie für unterschiedliche Wartebedingungen auch unterschiedliche Waitqueues einsetzen. Das vermeidet unnötiges Aufwecken von Rechenprozessen und spart Rechenzeit.
Software muss immer exakt erledigen, was der Anwender von ihr erwartet, und dabei fehlerfrei arbeiten. Sie darf keine unerwarteten Aktionen ausführen. Wer diese Grundregeln befolgt, wird nicht – wie bei Excel geschehen [3] – einen funktionierenden Flugsimulator in seinem Code verstecken. *(fjl)* ■

Listing 1: Private Daten

```
01 int driver_open (
02   struct inode *geraetedatei,
03   struct file *instanz)
04 {
05   struct p_instanz *priv;
06
07   priv = (struct p_instanz *) kmalloc
08     (sizeof(struct p_instanz), GFP_USER);
09   [...]
10   instanz->private_data = (void *) priv;
11   [...]
12 }
```

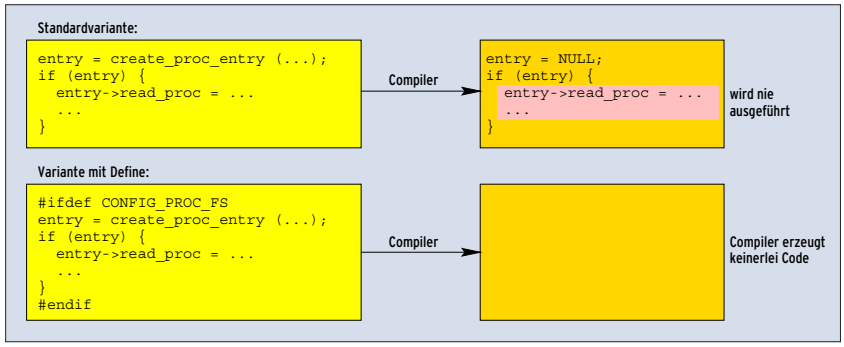

Abbildung 3: Das Ausmaskieren optionaler Teile per »#define« ist zwar nicht zwingend nötig, spart aber Code. Die Unterstützung für das Proc-Filesystem ist beide Male im Kernel deaktiviert. Im oberen Fall muss der Compiler dennoch Code erzeugen. Die Define-Variante vermeidet dagegen unnötige Funktionsaufrufe.

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik, Folge 5“: Linux-Magazin 12/03, S. 82
[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik, Folge 4“: Linux-Magazin 11/03, S. 96
[3] In Excel 97 ist ein Flugsimulator versteckt: [http://www.mogelpower.de/easter/eggs/egg.php?id=58]

Die Autoren

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

Do-While-0-Makros

Im Kernel tauchen häufig Makros auf, die in einer »do {...} while(0)«-Schleife verpackt sind. Auffällig daran: Die Bedingung ist fest vorgegeben und kann nie wahr sein, der Schleifenblock läuft also genau einmal ab. [http://www.kernelnewbies.org/faq/#dowhile] erklärt den Trick: So lassen sich Makros, die aus mehreren Anweisungen bestehen, gefahrlos in einer If-Anweisung verwenden. Einfacher wäre es, die Kommandos als Block zusammenzufassen:

```
#define FOO(x) { \
  pr_debug("error on input %d\n", x); \
  return -1; }
```

Das führt zu Fehlern, wenn ein Programmierer das Makro in einer If-Else-Anweisung nutzt:

```
if (fd<0)
  FOO(fd);
else
  something_else(fd);
```

Der C-Präprozessor ersetzt »FOO(fd)« durch das Makro, dabei entsteht aber ungültiger Code:

```
if (fd<0)
  {
  pr_debug("error on input %d\n", fd);
  return -1; };
else
  something_else(fd);
```

Problematisch ist das Semikolon nach dem eingefügten Block. Der Compiler interpretiert es als leeres Statement und koppelt den Else-Zweig ab. Der Programmierer hätte nach »FOO(fd)« kein Semikolon schreiben dürfen. Dazu müsste er aber wissen, wie »FOO()« implementiert ist – wäre es eine Funktion oder ein Makro, das nur eine Anweisung einfügt, wäre das Semikolon wieder erforderlich.

Im Dienste der Codequalität

Eine Do-While-0-Schleife umgeht das Problem recht elegant. Der Präprozessor fügt damit genau eine Anweisung ein, der Programmierer braucht sich daher nicht um die Interna des Makros zu sorgen:

```
#define FOO(x) do { \
    pr_debug("error on input %d\n", x); \
    return -1; \
  } while(0)
```

Eingesetzt in das If-Else-Statement ergibt sich fehlerfreier Code:

```
if (fd<0)
  do {
    pr_debug("error on input %d\n", fd);
    return -1;
  } while(0);
else
  something_else(fd);
```

Auf die Laufzeit hat dieses Konstrukt keinen negativen Einfluss, da gute Compiler die unnötige Do-While-Schleife wegoptimieren.