

Praktikum angewandte Systemsoftwaretechnik (PASST)

Kernel-Module / Aufgabe 5

24. Mai 2018

Stefan Reif, Peter Wägemann, Florian Schmaus, Michael Eischer,
Andreas Ziegler, Bernhard Heinloth und Benedict Herzog

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Kernel-Module

Großteil des Codes im Linux-Kern besteht aus Gerätetreibern:

- Kernkomponenten (Prozessor, IRQ-Controller, Timerbausteine, ...)
 - Werden **immer** für ein funktionales System benötigt
 - Unmittelbarer Teil des Kerns
- Peripheriegeräte
 - Bussysteme: PCI(e), SATA, USB, ...
 - Treiber für einzelne Geräte und Geräteklassen: Tastatur, Maus, Grafikkarten, Festplatten, Soundkarten, ...
- Abhängig von der Hardware werden Module für die Geräte bei Bedarf geladen

Großteil des Codes im Linux-Kern besteht aus Gerätetreibern:

- Kernkomponenten (Prozessor, IRQ-Controller, Timerbausteine, ...)
 - Werden **immer** für ein funktionales System benötigt
 - Unmittelbarer Teil des Kerns
- Peripheriegeräte
 - Bussysteme: PCI(e), SATA, USB, ...
 - Treiber für einzelne Geräte und Geräteklassen: Tastatur, Maus, Grafikkarten, Festplatten, Soundkarten, ...
- Abhängig von der Hardware werden Module für die Geräte bei Bedarf geladen

Aufgabe 5

Entwicklung eines Kerneltreibers für ein USB-Gerät

- Dokumentation (ja, es gibt sie!)
 - Documentation/ enthält Anleitungen, Erklärungen, Beschreibung von Konzepten für die verschiedensten Teile des Linux-Kerns

Entwickeln im Linux-Kern

- Dokumentation (ja, es gibt sie!)
 - `Documentation/` enthält Anleitungen, Erklärungen, Beschreibung von Konzepten für die verschiedensten Teile des Linux-Kerns
- Zusätzlich kann man für große Teile des Linux-Kerns eine Beschreibung der Interfaces ähnlich Doxygen generieren

```
01 $ cd <KERNEL_SOURCES>  
02 $ make htmldocs
```

... landet in `Documentation/output/`

Entwickeln im Linux-Kern

- Dokumentation (ja, es gibt sie!)
 - `Documentation/` enthält Anleitungen, Erklärungen, Beschreibung von Konzepten für die verschiedensten Teile des Linux-Kerns
- Zusätzlich kann man für große Teile des Linux-Kerns eine Beschreibung der Interfaces ähnlich Doxygen generieren

```
01 $ cd <KERNEL_SOURCES>  
02 $ make htmldocs
```

... landet in `Documentation/output/`

- Für beides gilt: Always take with a grain of salt
Linux hat keine stabile API innerhalb des Kerns
 - Dokumentation kann veralten
 - sich auf eine alte Version des Interfaces beziehen
 - oder schlichtweg falsch sein
- Die beste Dokumentation ist oft der Code von anderen

- Die meisten Geräte können mehrfach vorhanden sein
 - Daten für die Instanzen müssen dynamisch allokiert werden
 - Beim Entfernen des Gerätes muss man sie dynamisch wieder freigeben
- Dynamische Speicherverwaltung - wie macht man das im Kern?
 - `malloc` und `free` funktionieren im Linux-Kern nicht einfach so
 - Dafür gibt es eine eigene API: `kmalloc()`, `kzalloc()`, `kfree()`

Entwickeln im Linux-Kern (2/2)

- Wie unterscheidet sich Kernel-Code sonst noch von Userlevel-Code?

[Documentation/output/kernel-hacking/index.html](#)

Guter Einstieg in die Kernel-Entwicklung

Liefert eine Übersicht über die Besonderheiten der Entwicklung von Kernel-Code

[Documentation/output/core-api/kernel-api.html](#)

Enthält eine Interfacebeschreibung für viele

Kernkomponenten und Bibliotheken (u.a. ein Subset der C-Bibliothek)

Ein einfaches Kernelmodul

```
01 #include <linux/module.h>
02 #include <linux/kernel.h> /*printk*/
03
04 int __init simple_module_init(void){
05     printk("module loaded\n");
06 }
07 void __exit simple_module_exit(void){
08     printk("module unloaded\n");
09 }
10 module_init(simple_module_init);
11 module_exit(simple_module_exit);
12
13 MODULE_LICENSE("GPL");
```

Gerätetreiber in Linux – Module (2/2)

■ Makefile

```
01 obj-m += simple_module.o
02
03 all:
04     make -C <KERNEL_SOURCE> M=$(PWD)
05
06 clean:
07     make -C <KERNEL_SOURCE> M=$(PWD) clean
```

■ Kann man einfach laden

```
01 $ insmod simple_module.ko
```

■ ... und entladen

```
01 $ rmmod simple_module
```

Hardware – Universal Serial Bus (USB) (1/2)

- Asymmetrischer Bus (Baum)
 - Ein Host (PC) (Wurzelknoten) und viele Functions (angeschlossene Geräte, Blätter)
 - Kommunikation wird ausschließlich vom Host initiiert
 - Geräte können nicht autonom miteinander kommunizieren
- Geschwindigkeitsstufen
 - Low-Speed (1,5 Mbit/s), USB 1.0
 - Full-Speed (12 Mbit/s), USB 1.0
 - High-Speed (480 Mbit/s), USB 2.0
 - SuperSpeed (5 Gbit/s), USB 3.0
 - SuperSpeedPlus (10 Gbit/s), USB 3.1 Gen 2

Hardware – Universal Serial Bus (USB) (2/2)

- Vier unterschiedliche Kommunikationsmechanismen:
 - Control Transfers** Unregelmäßige Anfragen vom PC an das Gerät
z.B. Enumeration Sequence
 - Bulk Transfers** Aperiodisch; für große Pakete ohne zeitliche Garantien
z.B. USB-Storage-Device
 - Interrupt Transfers** Periodische Kommunikation; begrenzte Antwortzeit
z.B. Maus, Tastatur
 - Isochronous Transfers** Periodische, kontinuierliche Datenströme
z.B. Webcam

Gerätetreiber in Linux - USB-Geräte

- Tiefere Ebenen des USB-Protokolls sind in Form eines Host-Controller-Treibers (HCD) schon implementiert
- Benutzung der unterschiedlichen USB-Transferarten direkt möglich
- Diese Funktionalität kann über `<linux/usb.h>` eingebunden werden
- Writing USB Device Drivers
 - Registrieren eines USB-Gerätetreibers im System
 - Anschließen und Entfernen von USB-Geräten
 - Kommunikation mit dem Gerät
 - Asynchrone USB-Transfers per **USB Request Blocks (URB)**
 - Synchrone USB-Transfers für die Aufgabe ausreichend

`Documentation/output/driver-api/usb/
writing_usb_driver.html`

USB: Endpoints und Pipes

- USB-Geräte bieten Kommunikationsendpunkte (Endpoints) an
- Auf Hostseite spricht man mit einem Gerät über einen Kanal (Pipe), der mit einem bestimmten Endpunkt (Endpoint) verbunden ist
- Art und Anzahl der Endpunkte sind gerätespezifisch
- Alle Geräte müssen mindestens den Endpunkt o bereistellen, der für Control Transfers benötigt wird (u.a. für die Konfiguration)

Weitere Informationen

- USB Spezifikation (siehe /proj/i4passt/doc)
- <http://www.beyondlogic.org/usbnutshell/usb1.shtml>

USB-Temperatursensor

- Bauanleitung und Quellen
 - Firmware & Userspacetreiber:
<http://www.poempelfox.de/ds1820tousb/>
 - neues Layout:
<https://gitlab.cs.fau.de/i4/passtboard-v2>
- Steuert mehrere Temperatursensoren über 1-Wire-Bus an
- Steuerung vom PC aus per USB Control Transfers möglich
 - Rescan der angeschlossenen Temperatursensoren
 - Temperatur- und Statusinformationen der einzelnen Sensoren auslesen
 - Reset des kompletten Gerätes
 - Synchroner Transfers für die Aufgabe ausreichend

- Abwicklung über den immer vorhandenen Endpunkt 0
- **Festverdrahtete** (Konfiguration etc.) und **gerätespezifische** Befehle
- Parameter für Control Transfers

Parameter	Größe	Beschreibung
request type	1 Byte	Charakteristik der Anfrage
request value	1 Byte	Nummer der Anfrage
index	2 Byte	1. Parameter für die Anfrage
length	2 Byte	2. Parameter für die Anfrage
		Länge des Datenpaketes

(vgl. USB Spezifikation 9.3)

Befehle für den Temperatursensor

- Der **Request-Type** für die Befehle ist immer gleich (USB Spec S. 248):
 - Datentransferrichtung ist vom Gerät zum PC
 - Anfragen sind vendor-spezifisch
 - Ziel der Anfrage ist das Gerät
request type 0xc0
- Kurze Statusabfrage:

Aufrufparameter

```
01 request      1
02 value        0
03 index        0
```

Antwort

```
01 struct short_status {
02     uint8_t  version_high;
03     uint8_t  version_low;
04     uint32_t timestamp;
05     uint8_t  supported_probes;
06     uint8_t  padding;
07 }__packed;
```

- supported_probes: Über die Lebenszeit des Gerätes am Bus konstant

Befehle für den Temperatursensor

■ Lange Statusabfrage:

Aufrufparameter

```
01 request      3
02 value        0
03 index        0
```

Antwort

```
01 struct probe_status {
02     uint8_t  serial[6];
03     uint8_t  type;
04     uint8_t  flags;
05     uint8_t  temperature[2];
06     uint32_t timestamp;
07     uint8_t  padding[2];
08 }__packed;
09 struct probe_status
10     answer[supported_probes];
```

- Liefert immer Status für alle unterstützten Sensoren
- Flags: **0x01** Sensor ist vorhanden, sonst ist der Slot unbenutzt
0x02 Sensor wird parasitär mit Spannung versorgt
- Mehrere Bytes umfassende Werte sind little-endian
- Temperatur ist ein 12-bit Zweierkomplement-Wert in sechzehntel Grad (für ds18b20 mit type == 0x28)

Befehle für den Temperatursensor

- Neuerkennung aller Sensoren am 1-Wire-Bus:

Aufrufparameter

```
01 request      2
02 value        0
03 index        0
```

Antwort

```
01 struct rescan_reply {
02     uint8_t  answer;
03 };
```

- Im Erfolgsfall zwei Antworten möglich
 - 23 Neuerkennung wird gestartet
 - 42 Neuerkennung wird schon durchgeführt

Befehle für den Temperatursensor

- Neuerkennung aller Sensoren am 1-Wire-Bus:

Aufrufparameter

```
01 request    2
02 value      0
03 index      0
```

Antwort

```
01 struct rescan_reply {
02     uint8_t  answer;
03 };
```

- Im Erfolgsfall zwei Antworten möglich
 - 23 Neuerkennung wird gestartet
 - 42 Neuerkennung wird schon durchgeführt
- Reset des kompletten Gerätes:

Aufrufparameter

```
01 request    4
02 value      0
03 index      0
```

- Das Geräte sollte bei diesem Kommando keine Antwort schicken
- Das Bereitstellen eines Empfangspuffers schadet trotzdem nicht

sysfs - Kernelzustand für Benutzer sichtbar machen

- Interaktion mit dem USB-Gerät via `sysfs`
- Benutzung von `sysfs`:
Documentation/filesystems/sysfs.txt
- In a Nutshell
 - Große Teile des Kerns sind aus `kobjects` aufgebaut
 - Objektorientierung in C: Documentation/kobject.txt
 - Struktur von `sysfs` spiegelt die Objektstruktur im Kern wieder
 - `kobjects` erscheinen im `sysfs` als Verzeichnis
 - Erzeugen von Dateien durch
`sysfs_create_file(&kobject, attr)`
 - Löschen von Dateien mit
`sysfs_remove_file(&kobject, attr)`
 - Es gibt auch Schnittstellen speziell für Geräte(-Treiber), siehe `sysfs`-Dokumentation

sysfs-Einträge sollen folgende Funktionen bereitstellen:

- Temperatur jedes Sensors durch Lesen einer eigenen Datei abfragbar

```
01 $ ls /sys/bus/usb/devices/4-1.5:1.0/  
02 bInterfaceNumber bNumEndpoints modalias power ...  
03 temp0 temp1 temp2 rescan reset  
04 $ cat /sys/.../temp1  
05 23.43
```

- Rescan des 1-Wire-Bus
„Hotplug“ von Sensoren auf der Platine

```
01 $ echo 1 > rescan
```

- Reset des Gerätes

```
01 $ echo 1 > reset
```

Reale USB-Geräte an eine KVM weiterleiten:

- Ein bestimmtes Gerät
 - usbdevice host:bus.addrMühsam wegen Hotplug an verschiedenen Ports:
Bus- und Adress-ID nicht zwingend eindeutig
- Ganze Geräteklassen
 - usbdevice host:vendor_id:product_idFür unsere Temperatursensoren ist das 16c0:05dc
 - usbdevice host:16c0:05dc
- *Problem:* KVM benötigt Lese- und Schreibrechte fürs Gerät
Lösung: udev

```
01 ATTRS{idVendor}=="16c0", ATTRS{idProduct}=="05dc", MODE="666"
```

```
/etc/udev/rules.d/99-usbtemp.rules
```

Userspace-Anwendung zur Darstellung der Messdaten

- Temperaturen periodisch aus `/sys` auslesen
- evtl. geeignet Zwischenspeichern
- zeitlichen Verlauf aller Sensoren in einem Graphen ausgeben
- mehr Punkte wenn der Graph „live“ aktualisiert wird
- mögliche Werkzeuge u.A.:
 - GNUPLOT
 - RRDTOOL
 - R
 - ROOT (`root.cern.ch`)

Aufgabe 5

USB-Temperatursensor

- Einarbeiten in die benötigten APIs im Linux-Kern
 - Dokumentation
 - Codebeispiele
- Programmieren des Gerätetreibers für den Temperatursensor
- Programmieren der Userspace-Anwendung
- Last, but not least:
 - Die Hardware muss gelötet werden

Abgabe: bis 2018-07-06 durch Vorführung in einer Rechnerübung

Fragen?