Fachprojekt „Systemsoftwaretechnik”

03 - Der eigene Treiber

Alexander Krause

AG Systemsoftware

Veranstaltungswebseite

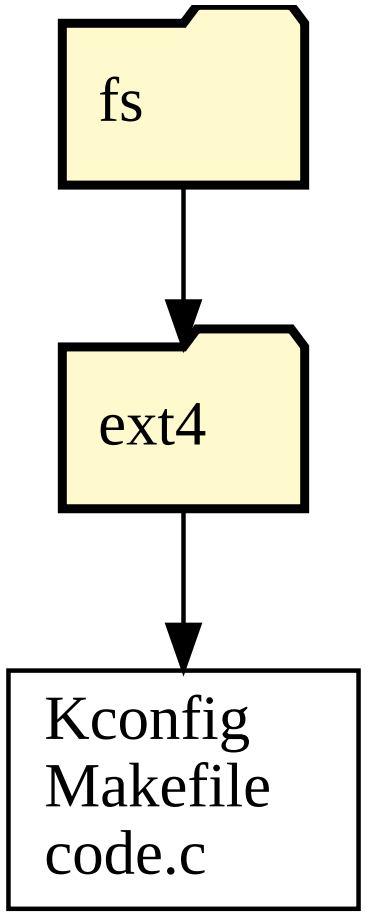
11.06.2024

Wie baue ich ein neues Feature ein?

- Grundstruktur anlegen
- Konfiguration erstellen
- Code schreiben

Grundstruktur

Was benötige ich alles für meinen Treiber?



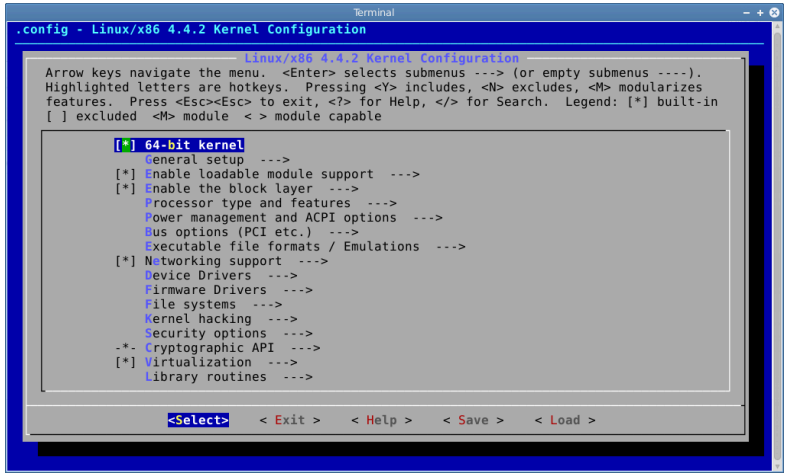
- Verzeichnis: `fs/ext4`
- Beschreibung, was gebaut werden soll: `fs/ext4/Kconfig`
- Beschreibung, was alles dazugehört: `fs/ext4/Makefile`
- Natürlich den Quellcode selbst: `fs/ext4/code.c`

Wie baue ich ein neues Feature ein?

- Grundstruktur anlegen ✅
- Konfiguration erstellen
- Code schreiben

Konfiguration

Kconfig (Wdh.)



- Konfigurationssprache des Linux-Kerns
- Verwaltet Konfigurationsoptionen zur Übersetzungszeit
- Berücksichtigt Abhängigkeiten und setzt diese durch
- Bedeutung der Symbole
  - < > keine Abhängigkeiten
  - [ ] kann einkompiliert (*y*) werden oder nicht (*n*)
  - { } als Modul (*m*) oder einkompiliert (*y*) benötigt
  - - - einkompiliert (*y*) benötigt

Kconfig unter der Haube

```
config EXT4_FS
    tristate "The Extended 4 (ext4) filesystem"
    select JBD2
    select CRC16
    select CRYPTO
    select CRYPTO_CRC32C
    select FS_IOMAP
    select FS_ENCRYPTION_ALGS if FS_ENCRYPTION
    help
      This is the next generation of the ext3 filesystem.
      [...]
      If unsure, say N.
```

(Aus fs/ext4/Kconfig)

- `config` leitet eine Konfigurationsoption ein, dahinter der Bezeichner der Option
- `tristate` bezeichnet den Typ der Option (`tristate`, `bool`, `int`, `string`, ...)
- `select` wählt zusätzliche Optionen, wenn diese Option aktiviert wird
- `help` ist selbst erklärend
- Fehlt: `depends` → Beschreibt Abhängigkeiten zu anderen Optionen
- Konfigurationsoptionen stehen sowohl im Quellcode als auch im Makefile mit dem Prefix `CONFIG_` zur Verfügung

Von der Konfiguration zum Quellcode – Grundlagen

- Makefile-Variablen legen Art der Übersetzung fest (siehe 3. in Dokumentation)
  - `obj-y += foo.o`: Übersetze Datei `foo.c` immer
  - `obj-m += foo.o`: Übersetze Datei `foo.c`, wenn die Modulunterstützung aktiv ist
- Konfigurierbare Übersetzung
  - `obj-$(CONFIG_FOO) += foo.o`
  - Übbersetze `foo.c`, wenn Konfigurationsoption `CONFIG_FOO` entweder *y* oder *m* ist
- Binde Unterverzeichnisse ein

```
obj-$(CONFIG_EXT4_FS)       += ext4/
```

(Aus fs/Makefile)

Von der Konfiguration zum Quellcode – *Makefile*

```
1  # SPDX-License-Identifier: GPL-2.0
2  #
3  # Makefile for the linux ext4-filesystem routines.
4  #
5
6  obj-$(CONFIG_EXT4_FS) += ext4.o
7
8  ext4-y  := balloc.o bitmap.o block_validity.o dir.o ext4_jbd2.o extents.o \
9          extents_status.o file.o fsmap.o fsync.o hash.o ialloc.o \
10         indirect.o inline.o inode.o ioctl.o mballoc.o migrate.o \
11         mmp.o move_extent.o namei.o page-io.o readpage.o resize.o \
12         super.o symlink.o sysfs.o xattr.o xattr_hurd.o xattr_trusted.o \
13         xattr_user.o fast_commit.o orphan.o
14
15 ext4-$(CONFIG_EXT4_FS_POSIX_ACL)    += acl.o
16 # [....]
17 obj-$(CONFIG_EXT4_KUNIT_TESTS)      += ext4-inode-test.o
18 # [....]
```

(Aus fs/ext4/Makefile)

- Modul `ext4.o` besteht aus mehreren Quellcodedateien
- Umweg über separate Variable gemäßg Modulname: `ext4-y :=`
- Syntax ist genauso wie bereits erwähnt

Wie baue ich ein neues Feature ein?

- Grundstruktur anlegen ✅
- Konfiguration erstellen ✅
- Code schreiben

Gerätetreiber

Von Systemaufruf zum Treiber

```c
char buf[42];

int fd = open("/dev/the-universe", O_RDWR);
if (read(fd, buf, 42)) {
    // ...
}
```

```
(gdb) bt
#0  universe_read (file=0xffff88800453b700,
    buf=0x7f9b78c4a000 <error: Cannot access memory at address 0x7f9b78c4a000>,
    count=42, ppos=0xffffc90000507ef0) at drivers/sst/sst_chrdev.c:25
#1  0xffffffff812a5579 in vfs_read (file=file@entry=0xffff88800453b700,
    buf=buf@entry=0x7f9b78c4a000 <error: Cannot access memory at address 0x7f9b78c4a000>,
    count=count@entry=131072, pos=pos@entry=0xffffc90000507ef0) at fs/read_write.c:468
#2  0xffffffff812a60e3 in ksys_read (fd=<optimized out>,
    buf=0x7f9b78c4a000 <error: Cannot access memory at address 0x7f9b78c4a000>,
    count=131072) at fs/read_write.c:613
#3  0xffffffff812a6179 in __do_sys_read (count=<optimized out>, buf=<optimized out>,
    fd=<optimized out>) at fs/read_write.c:623
#4  __se_sys_read (count=<optimized out>, buf=<optimized out>, fd=<optimized out>)
    at fs/read_write.c:621
#5  __x64_sys_read (regs=<optimized out>) at fs/read_write.c:621
#6  0xffffffff81b1b055 in do_syscall_x64 (nr=<optimized out>, regs=0xffffc90000507f58)
    at arch/x86/entry/common.c:50
#7  do_syscall_64 (regs=0xffffc90000507f58, nr=<optimized out>)
    at arch/x86/entry/common.c:80
#8  0xffffffff81c0006a in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:120
```

Zustellen von Systemaufrufen zu Treibern

```
1  ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
2  {
3      ssize_t ret;
4
5      if (!(file->f_mode & FMODE_READ))
6          return -EBADF;
7      if (!(file->f_mode & FMODE_CAN_READ))
8          return -EINVAL;
9      if (unlikely(!access_ok(buf, count)))
10         return -EFAULT;
11
12     //  [...]
13     if (file->f_op->read)
14         ret = file->f_op->read(file, buf, count, pos);
15     else if (file->f_op->read_iter)
16         ret = new_sync_read(file, buf, count, pos);
17     else
18         ret = -EINVAL;
19     if (ret > 0) {
20         fsnotify_access(file);
21         add_rchar(current, ret);
22     }
23     inc_syscr(current);
24     return ret;
25 }
```

(Aus fs/read_write.c:468)

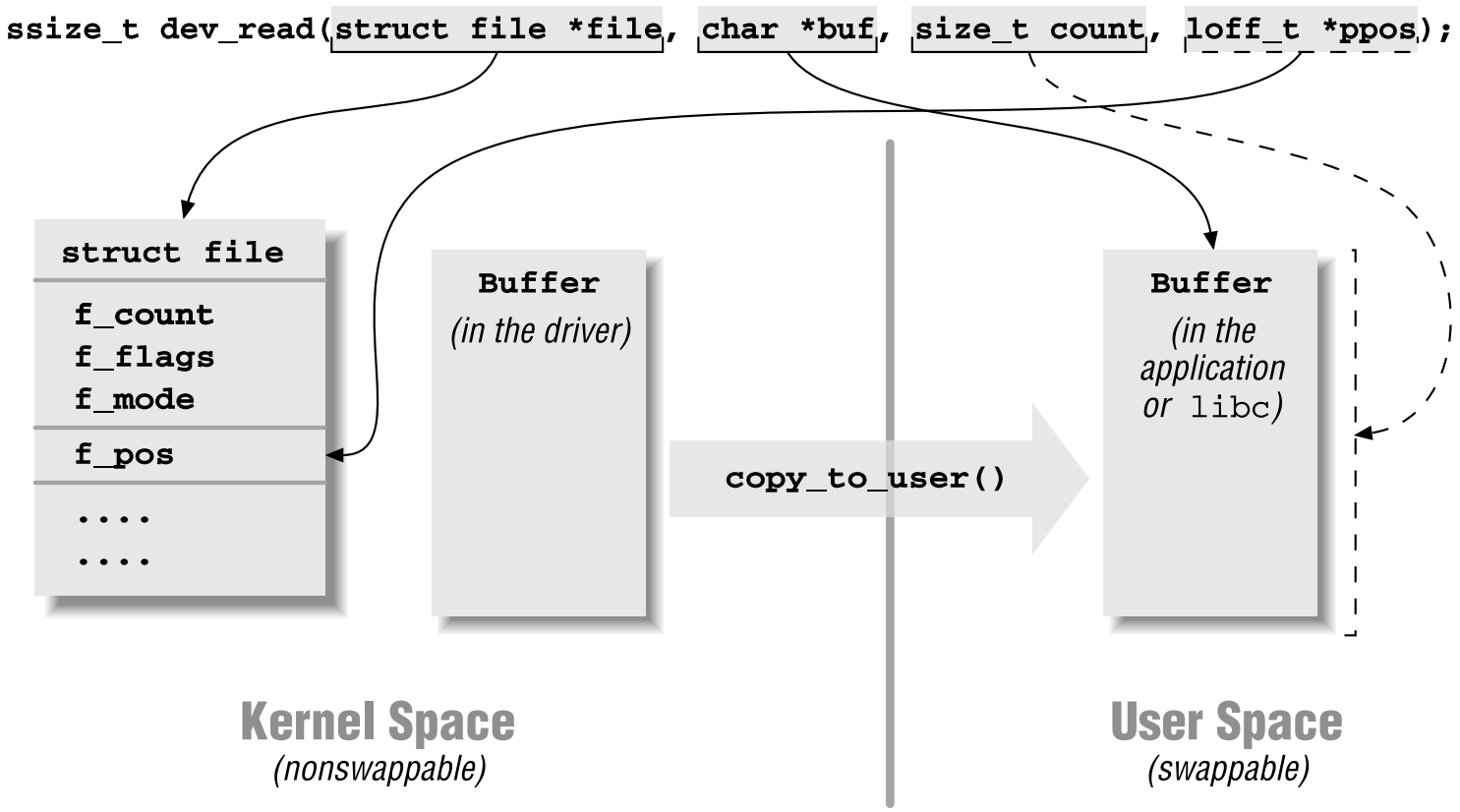
Polymorphie in C 😖

- `UNIX`-Paradigma
  - „Alles ist eine Datei”
  - Dateioperationen definieren Interaktionen: *open*, *read*, *write*, *lseek*, ..., *close*
- Schnittstellendefinition über Funktionszeiger
- Definition der Schnittstelle in `struct file_operations`:

```
1  struct file_operations {
2     struct module *owner;
3     loff_t (*llseek) (struct file *, loff_t, int);
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7      // [...]
8  } __randomize_layout;
```

(Aus include/linux/fs.h:2109)

Was bedeuten die Parameter beim *read*-Systemaufruf?



Datenstruktur hinter Dateideskriptor – `struct file`

```
1  struct file {
2      // [...]
3     struct path             f_path;
4     struct inode            *f_inode;       /* cached value */
5     const struct file_operations    *f_op;
6      // [...]
7     unsigned int            f_flags;
8     fmode_t                 f_mode;
9     struct mutex            f_pos_lock;
10    loff_t                  f_pos;
11     // [...]
12    /* needed for tty driver, and maybe others */
13    void                    *private_data;
14     // [...]
15 } __randomize_layout
16   __attribute__((aligned(4)));     /* lest something weird decides that 2 is OK */
```

(Aus include/linux/fs.h:940)

- Repräsentiert im Kern eine geöffnete Datei
- Weitere Informationen in „Linux Device Driver” (*Buch*seite 66 [1])

Die Programmierschnittstelle (*API*)

Ausgaben

- Grundsätzliches Vorgehen ist wie bei `printf()` (Formatstring, Argumente, …)
- Pendant im Kernel heißt `printk()`, Semantik ist aber gleich!
- Makros für das passende Log-Level: `pr_err`, `pr_info`, `pr_debug`, …
- `printk_ratelimited()` 🤔
- Definition und (etwas) Dokumentation in include/linux/printk.h:434
- Ergänzungen zu `pr_debug`:

```
1 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
2 #include <linux/printk.h>
3 #define sst_debug(format, ...) pr_debug("(file:%s,line:%d,pid:%d): " format,       \
4             __FILE__, __LINE__,  current->pid, ##__VA_ARGS__)
```

Dynamische Speicherverwaltung (im Linux-Kern)

- `vmalloc/vfree`
  - Allokation von **virtuell** adjazentem Speicher
  - Bekommt Größe übergeben
- `kmalloc/kfree`
  - Allokation von **physikalisch** adjazentem Speicher
  - Bekommt Größe und Flags übergeben – siehe Dokumentation
- `kmalloc-Flags`
  - `GFP_KERNEL`: „Allocate normal kernel ram. May sleep.”
  - `GFP_NOWAIT`: „Allocation will not sleep.”
  - `GFP_ATOMIC`: „Allocation will not sleep. May use emergency pools.”

Kernel-Threads

- Kernel-Threads sind „standard processes that exist solely in kernel-space” (*Buch*seite 35, [2])
- Verhalten sich wie normale Prozesse: unterliegen Ablaufplanung und Präemption
- Wichtiger Unterschied: verfügen über keinen keinen eigenen Adressraum, laufen im Adressraum des Kerns
- Erzeugung läuft intern auch über Systemaufruf `clone()`, wie beim Systemaufruf `fork()`
- Bei Erzeugung nicht lauffähig; müssen explizit gestartet werden

Kernel-Threads – API

- Thread erstellen:

```
struct *task_struct kthread_create(int (*threadfn)(void
*data), void
```

- Erstellten Thread a

```
int wake_up_p
```

- Warten bis der Thread sich beendet:

```
int kthread_stop(struct task_struct *tsk)
```

- Soll der Thread anhalten?

```
bool kthread_should_stop(void)
```

Weitere Funktionen und „Dokumentation” finden sich in include/linux/kthread.h

Kernel-Threads – Beispiel

```c
#include <linux/kthread.h>

char *text = "Hello from the other side!";
struct task_struct *faden;

static int work_fn(void *arg) {
    char *text = (char*)arg;

    while (!kthread_should_stop()) {
        printk("Working: %s\n", text);
        ssleep(1);
    }
    return 0;
}

static init __init module_init(void) {
    faden = kthread_create(work_fn, (void*)text, "ein-name");
    if (IS_ERR(faden)) {
        pr_err("Error\n");
    }
    return 0;

    wake_up_process(faden);
    return 0;
}

// kthread_stop(faden);
```

Synchronisation

- Ganze Code-Abschnitte
  - (RW-)Spinlocks
  - Mutex
  - (RW-)Semaphore
  - Sequential Locks
  - Abschalten von Präemption/Unterbrechungen
- Auf Instruktionsebene
  - Atomare Operationen
  - Barrieren
- Sonstige
  - *Completion Variables*

(Siehe Kapitel 10 in „Linux Kernel Development” [2])

Synchronisation – API

- Spinlocks
  - `spin_lock/spin_unlock`
  - Abschalten der Softirqs bzw. Unterbrechungen mit Suffix: `_bh`, `_irq`, `_irqsave`
- Mutex/Semaphore
  - `down/up` (Besser: `down_interruptible`)
  - `mutex_lock/mutex_unlock`
- Vergleich zwischen Spinlocks und Mutex/Semaphore in Kap. 10, Seite 197 [2]
- Verwendung eines Semaphores siehe `drivers/sst/sst_common.c` in Zeile 146 bzw. 171)

Kernel Library

- Kern-weite Bibliothek
  - Circular Buffers (oder Ihr nehmt den `BoundedBuffer` aus unserem Treiber. Fragt gerne nach.)
  - Zufallszahlen: `get_random_uX()` – siehe include/linux/random.h
  - Operationen für Zeichenketten: `strcpy()`, `strcmp()`, … (Dokumentation)
  - Einlesen von Zahlen: `kstrtol()` (Dokumentation) oder `sscanf()` (Dokumentation)
  - Verkettete Listen (*Buch*seite 85 ff. [2] und im Code)

Bounded Buffer

- Dürft Ihr gerne verwenden.
- Implementierung liegt in `drivers/sst/boundedbuffer.{c,h}`

```
1  // Größe beträgt 20 Elemente
2  #define BOUNDEDBUFFER_SIZE 20
3  // Gespeichert werden Zeiger auf einen Char
4  #define BOUNDEDBUFFER_STORAGE_TYPE char*
5  #include "boundedbuffer.h
6
7  // [...]
8
9  struct boundedbuffer foo;
```

Fehlerbehandlungen

- Beachtet **immer** den Rückgabewert einer Funktion
- Behandelt **immer** den Fehlerfall
- Macht bereits abgeschlossenen Teilschritte **immer** rückgängig
- Fehler sind im Kern sind besonders fatal

→ Im Zweifel eine Ausgabe und einen Fehlercode zurückgegeben

```
1  int init_pferd(void) {
2      foo = alloc_foo();
3
4      spin_lock(&a_lock);
5      if (!init_bar()) {
6          // Was muss hier alles passieren?
7      }
8      spin_unlock(&a_lock);
9
10     return 0;
11 }
```

Wie baue ich ein neues Feature ein?

- Grundstruktur anlegen ✅
- Konfiguration erstellen ✅
- Code schreiben ✅

Dokumentation

Dokumentation

- Quellcode und Kommentare, sofern vorhanden. 😱 😍 🥴 🤮
- Kern-Dokumentation
- Bücher
  - „Linux Kernel Development 3rd Edition” [2]
  - „Understanding The Linux Kernel” [3]
  - „Linux Device Drivers 3rd Edition” [1]
- Die Bücher habe ich alle im Büro liegen. 😎

Referenzen

[1]
J. Corbet, A. Rubini, und G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[2]
R. Love, *Linux Kernel Development*, 3rd Aufl. Addison-Wesley, 2010.

[3]
D. P. Bovet und M. Cesati, *Understanding The Linux Kernel*, 3rd Aufl. O'Reilly Media Inc., 2005.