

Betriebssystem-Entwicklung mit Literate Programming

Foliensatz 11: Dateisysteme



Wintersemester 2014/15

Hans-Georg Eßer

h.g.esser@cs.fau.de
<http://ohm.hgesser.de/>

v1.1, 16.12.2014

Dateisysteme (1)

- Frühe Betriebssysteme (CP/M, MS-DOS etc.):
 - ein einziges (logisches) Dateisystem, z. B. FAT
 - Support für wenige Gerätekategorien (z. B. nur Floppy-Laufwerke)
 - Code für Hardware-Zugriff und logische Behandlung des Dateisystems vermischt
- Moderne Betriebssysteme (Linux, Windows etc.):
 - virtuelles Dateisystem
 - getrennter Code für Hardware / Dateisysteme

Dateisysteme (2)

- Ulix setzt Methoden moderner BS ein
 - unterste Ebene: Hardware, einzelne Sektoren lesen, Funktionen
`readsector_hd()`, `writesector_hd()`,
`readsector_fd()`, `writesector_fd()`
 - oberste Ebene: virtuelles Dateisystem, Dateien und Verzeichnisse manipulieren, Funktionen
`u_open()`, `u_read()`, `u_write()`, `u_lseek()` etc.
 - Verbindungsstücke: Dateisystem-spezifische Funktionen, z. B. für Minix- oder FAT-Dateisystem

Dateisysteme (3)

- Ulix – So setzen sich die Komponenten zusammen:
 - Geräte über Geräte-IDs unterscheidbar (`DEV_HDA`, `DEV_HDB`, `DEV_FD0`, `DEV_FD1`), prinzipiell erweiterbar
 - File-Deskriptoren (für offene Dateien)
 - global (für BS und alle Prozesse)
 - verwaltet jedes Dateisystem selbst
 - getrennte Gruppen je Gerät

Ulix: VFS-Funktionen (1)

```
int u_open (char *path, int oflag) {
    /* Variablen-Deklarationen ausgelassen */

    // check relative/absolute path
    if (*path != '/') relpath_to_abspath (path, abspath);
    else                strncpy (abspath, path, 256);

    get_dev_and_path (abspath, ← VFS-Pfad
                    &device, &fs, &localpath); ← Pfad auf Gerät

    switch (fs) {
        case FS_MINIX:
            fd = mx_open (device, localpath, oflag);
            if (fd == -1) return -1; // error
            else return (fs << 8) + fd;

        case FS_FAT: return -1; // not implemented
        case FS_ERROR: return -1; // error
    }
}
```

Ulix: VFS-Funktionen (3)

- Die generische Funktion `u_open` weiß also nichts über den Aufbau von Minix-, FAT- oder sonstigen Dateisystemen
- Aber: kann rausfinden, welches Gerät und welches Dateisystem verwendet wird, und dann die richtige spezifische `open`-Funktion (z. B. `mx_open`) aufrufen

Ulix: VFS-Funktionen (2)

```
relpath_to_abspath (path, abspath);

z. B.: $PWD:    /home/ulix/
       path:    Documents/beispiel.txt

       abspath: /home/ulix/Documents/beispiel.txt

get_dev_and_path (abspath, &device, &fs, &localpath);

z. B.: Mount:    2. Floppy (FD1) auf /mnt gemountet,
                Typ minix
       abspath:   /mnt/tmp/beispiel.txt

       device:    DEV_FD1 (2. Floppy)
       fs:        FS_MINIX
       localpath: /tmp/beispiel.txt

mx_open (device, localpath, oflag);
```

Ulix: VFS-Funktionen (4)

- Jedes Subsystem (Minix, FAT etc.) verwaltet eigene File Descriptors (0, 1, 2, ...)
 - Auf VFS-Ebene werden diese zusammengesetzt, global eindeutig, z. B.
 - Minix: $FS_MINIX = 1$,
 - Minix-lokaler FD globaler FD
- | | |
|---|-------------------------------------|
| 0 | $1 \ll 8 + 0 = 256 + 0 = 256$ |
| 1 | $1 \ll 8 + 1 = 256 + 1 = 257$ |
| 2 | $1 \ll 8 + 2 = 256 + 2 = 258 \dots$ |

Ulix: VFS-Funktionen (5)

- Neben `u_open` zum Öffnen einer Datei sind weitere Standardfunktionen verfügbar, etwa
- `u_read`: aus Datei lesen
- `u_write`: in Datei schreiben
- `u_lseek`: an Position in Datei springen
- `u_close`: Datei schließen
- `u_unlink`: Datei löschen
- `u_mkdir`: Verzeichnis erzeugen etc.

Minix-Dateisystem (1)

Klassisches Unix-Dateisystem

- **Superblock**: enthält Verwaltungsinformationen über Datenträger
- **Inode**: Metadaten einer Datei (Größe, Datenblöcke, Zugriffsrechte etc.)
- **Inode-Bitmap**: speichert, welche Inodes verwendet / frei sind
- **Block-Bitmap**: speichert, welche Datenblöcke verwendet / frei sind

Ulix: VFS-Funktionen (6)

- Viele VFS-Funktionen haben einfachen Aufbau

```
int u_read (int fd, void *buf, int nbyte) {
    if (fd < 0) return -1; // file not open
    int fs      = fd >> 8;
    int localfd = fd & 0xff;

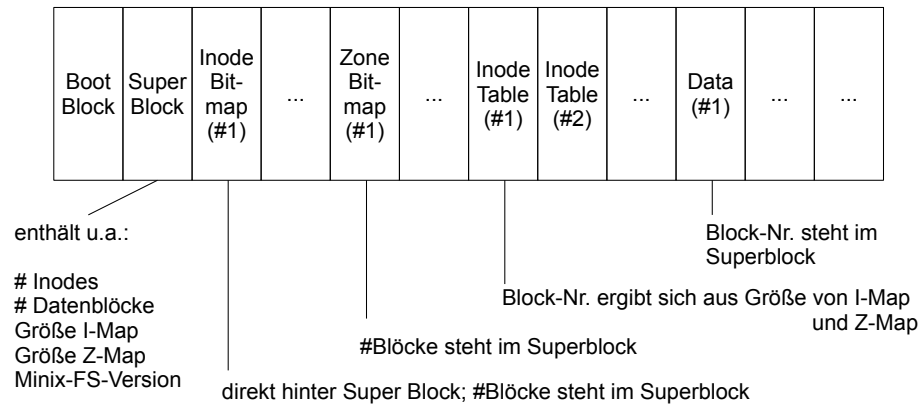
    switch (fs) {
        case FS_MINIX: return mx_read (localfd, buf, nbyte);
        case FS_FAT:   return -1; // not implemented
        case FS_ERROR: return -1; // error
    }
}
```

Minix-Dateisystem (2)

- Inodes enthalten *keinen* Dateinamen
- **Verzeichnisse**: spezielle Dateien, die Zuordnungen der Form
Dateiname → Inode-Nummer
enthalten

Minix-Dateisystem (3)

- Aufbau eines Minix-Dateisystems



Minix-Dateisystem (4)

- Minix-spezifische Funktionen (`mx_*`) verstehen den logischen Aufbau des Minix-FS
- Auf unterster Ebene werden schließlich einzelne 1-KB-Blöcke vom Datenträger angefordert:
- Aufruf von `readblock (device, block, buf)`
- `readblock` kann dann passende Geräte-spezifische Funktion aufrufen, z. B. `readblock_hd` oder `readblock_fd`

Blöcke lesen

- Beispiel `readblock`:
- Drei Geräte-Klassen (HD, FD, „Serial Disk“)

```
void readblock (int device, int blockno, char* buffer) {
    unsigned char major = device >> 8;    // obere 8 Bits
    unsigned char minor = device & 0xFF;  // untere 8 Bits
    switch (major) {
        case MAJOR_HD:    readblock_hd (minor/64, blockno, buffer);
                          break;
        case MAJOR_FD:    readblock_fd (minor,    blockno, buffer);
                          break;
        case MAJOR_RAM:   break; // not implemented
        case MAJOR_SERIAL: readblock_serial (    blockno, buffer);
                          break;
        default: break;
    }
}
```

DEV_HDA = 0x300 → major = 3, minor = 0
DEV_HDB = 0x340 → major = 3, minor = 0x40
DEV_FD0 = 0x200 → major = 2, minor = 0
DEV_FD1 = 0x201 → major = 2, minor = 1

Datentransfers

- Zugriff auf Datenträger immer nach gleichem Muster, Beispiel Lese-Operation:
 - gewünschte Blocknummer, Datenpuffer und Transferrichtung geeignet kodieren
 - Kommando mit `outport`-Befehlen an Controller schicken
 - schlafen legen (nach Abschluss kommt Interrupt)
 - *falls DMA*: alles schon fertig
 - *falls nicht DMA*: Daten aus Controller-Speicher kopieren (`inport`)

Beispiel: Auf Platte schreiben

```
void writesector_hd (int hd, int sector, char* buffer) {
    hd_direction = HD_OP_WRITE;

    // „controller magic“
    outportb (IO_IDE_DISKSEL, 0xe0 | (hd<<4)); // select disk
    outportb (IO_IDE_DEVCTRL, 0); // generate interrupt
    outportb (IO_IDE_SEC_COUNT, 1); // one sector
    outportb (IO_IDE_SECTOR, sector & 0xff);
    outportb (IO_IDE_SECTOR+1, (sector >> 8) & 0xff);
    outportb (IO_IDE_SECTOR+2, (sector >> 16) & 0xff);
    outportb (IO_IDE_SECTOR+3, ((sector >> 24) & 0x0f) | ((0xe + hd) << 4));
    outportb (IO_IDE_COMMAND, IDE_CMD_WRITE);

    inportb (IO_IDE_STATUS); // read status, ack irq
    repeat_outports1 (IO_IDE_DATA, buffer, HD_SECSIZE / 4);
    inportb (IO_IDE_STATUS); // read status, ack irq
    while (hd_direction == HD_OP_WRITE) {}; // wait for completion
    hd_direction = HD_OP_NONE;
}
```

(stark vereinfacht, ohne Prozesse)

Interrupt-Handler (ohne DMA)

```
void ide_handler (context_t *r) {
    switch (hd_direction) {
        case HD_OP_READ: <disable interrupts>
            // Daten vom Controller ins RAM kopieren
            repeat_inports1 (IO_IDE_DATA, hd_buf,
                HD_SECSIZE / 4);
            hd_direction = HD_OP_NONE;
            break;

        case HD_OP_WRITE: <disable interrupts>
            hd_direction = HD_OP_NONE;
            break;

        // wartenden Prozess aufwecken
        <enable interrupts>
    }
}
```

Interrupt-Handler

- Controller (HD oder FD) erzeugt einen Interrupt, wenn der Transfer abgeschlossen ist
 - **DMA:** Bei DMA ist der Interrupt das Signal dafür, dass der Transfer in den PC-Hauptspeicher abgeschlossen ist
 - **non-DMA:** Ansonsten liegen die Daten im internen Speicher des Controllers und müssen noch ausgelesen werden