

Ext4-Übung

Florian Mayer

30. Mai 2015

Inhaltsverzeichnis

1	Einführung	1
1.1	Bestandteile des e2fsprogs-Pakets	1
1.2	Verwendete Programme	2
2	Übung	2
2.1	Vorbereitungen des Test-Dateisystems	2
2.2	Allgemeine Fragen	3
2.3	Dumpe2fs	3
2.4	Einlesen des Superblocks	3
2.5	Anlegen von Testdaten	5
2.6	Debugfs	5
2.7	Inodes	6
3	Blockgruppen Layout von Ext4	7
4	Ext4-API-Referenz	7
4.1	Öffnen eines ext4-Dateisystems	7
4.2	Schließen eines Ext4-Dateisystems	8
4.3	Lesen eines Inodes	8

1 Einführung

Im ersten Kapitel dieses Dokuments werden die Werkzeuge des e2fsprogs-Pakets näher beleuchtet. Danach folgt die eigentliche Übung

1.1 Bestandteile des e2fsprogs-Pakets

Die Tools – soweit nicht anders vermerkt – arbeiten jeweils mit ext2, ext3 oder ext4 Dateisystemen. “ExtX” bezeichnet zusammenfassend ext2- ext3- oder ext4-Systeme.

e2fsk Überprüft ein ExtX-System auf Inkonsistenzen und behebt diese, wenn möglich.

mke2fs Wird vom Frontendprogramm **mkfs** verwendet um neue ExtX-Dateisysteme anzulegen.

resize2fs Kann genutzt werden, um ein ExtX-Dateisystem an eine gewachsene oder verkleinerte Partition anzupassen.

tune2fs Setzt oder modifiziert Dateisystemparameter.

dumpe2fs Schreibt Superblock- bzw. Blockgruppeninformationen auf die Standardausgabe.

filefrag Zeigt den Grad der Dateifragmentierung an.

e2label Verändert das Dateisystemlabel eines ExtX-Systems.

findfs Sucht nach einem Dateisystem mit einem Label oder einer UUID.

e2freefrag Wie **filefrag**, jedoch mit dem Unterschied, dass nur die Fragmentierung des freien Speicherplatzes aufgezeigt wird.

chattr Ähnlich zu **chmod** oder **chown**. Funktioniert auf vielen verschiedenen Dateisystemen.

e2image Speichert kritische ExtX-Metadaten in einer Datei.

e4defrag Defragmentiert ein Ext4-Dateisystem während es parallel weiterhin genutzt werden kann.

findsuper Findet Ext2-Superblocks. Veraltet und lt. Dokumentation "schnell zusammengehackt".

1.2 Verwendete Programme

Im Rahmen dieses Dokuments werden lediglich die Programme **debugfs** und **dumpe2fs** verwendet. Die Manualseiten könnten, wie gewohnt, mit

```
$ man debugfs
$ man dumpe2fs
```

angesehen werden. Das Studium dieser Handbücher ist dringend empfohlen.

2 Übung

2.1 Vorbereitungen des Test-Dateisystems

Operationen auf Dateisysteme mit sog. Debugging-Kommandos sind gefährlich. Löscht man z.B. versehentlich einen Inode, so ist es nur noch schwer möglich die zugehörige Datei wiederherzustellen. Es wird daher beschrieben, wie eine sog. Sandbox-Umgebung (also ein vom System abgekapseltes Dateisystem) eingerichtet werden kann. Die hier geschilderte Methode steht sofort unter GNU/Linux zur Verfügung, da sie integraler Kernel-Bestandteil ist.

- **dd if=/dev/zero of=~/.foo.disk count=1000**
Erzeugt eine genullte Datei, bestehend aus 1000 Blöcken zu jeweils 0,5KiB.
- **losetup --list**
Zeigt bereits vorhandene loop devices (footnote) an. Bereits vorhandene Loop-Geräte müssen mittels **sudo losetup -d /dev/loop<X>** ausgehängt werden.
- **sudo losetup /dev/loop0 ~/.foo.disk**
Erzeugt eine blockorientierte Gerätedatei, die alle Lese- und Schreiboperationen auf die Datei **~/.foo.disk** abbildet. Das Kommando muss sodann mithilfe von **losetup --list** überprüft werden. Der Vorgang war erfolgreich, wenn nun die neue Konfiguration ausgegeben wird.
- **sudo parted /dev/loop0 mklabel msdos**
Erstellt eine Partitionstabelle mit *msdos*-Format. Das bedeutet, dass ein normaler MBR erstellt wird.
- **sudo parted /dev/loop0 mkpart primary 0 500K**
Das erste Kommando erzeugt einen neuen MBR und das zweite richtet eine primäre Partition über die volle Länge des virtuellen Blockgeräts ein.

- `sudo mkfs.ext4 /dev/loop0p1`

Erzeugt das zu untersuchende Dateisystem. Achtung: Hier darf `/dev/loop0` nicht verwendet werden, da dies keine Partition ist! Die Gerätedatei der Partition (`/dev/loop0p1`) wird automatisch vom System erkannt und steht daher sofort nach Erzeugung der Partition zur Verfügung. Hinweis: Merken Sie sich die Ausgabe des Kommandos! Sie werden Sie in einer der Übungen benötigen!

- Nun muss das frische Dateisystem nur noch eingebunden werden:

```
$ sudo mkdir /mnt/foo
$ sudo mount /dev/loop0p1 /mnt/foo
```

2.2 Allgemeine Fragen

1. Ab welchem Offset beginnt der Superblock und wie groß ist er? Hinweis: Der Superblock beginnt typischerweise unmittelbar nach dem nullten Block, also im ersten (vgl. hierzu Kapitel 2.2 “Layout” [der offiziellen Dokumentation](#))
2. Wieviel Inodes und Datenblöcke wurden beim Erstellen des Dateisystems erzeugt? Hinweis: Die Ausgabe des `mkfs.ext4` Kommandos hilft. Alternativ kann auch `dumpe2fs` benutzt werden.
3. Wie groß ist ein Ext4-Datenblock und wie viele sind davon anfangs tatsächlich durch Dateien nutzbar? Hinweis: Nutzen Sie `dumpe2fs` und suchen Sie nach dem Feld “Block size”!
4. Wieviele Bytes ist ein Inode groß? Hinweis: Nutzen Sie wieder `dumpe2fs`

2.3 Dumpe2fs

Beantworten Sie mithilfe des Programms `dumpe2fs` die folgenden Fragen unter der Annahme, dass Sie sich auf einem *32-Bit-System* befinden¹. (Verwenden Sie für diese Übung Kapitel 2.1 [der offiziellen ext4-Dokumentation](#). Die relevanten Daten finden in der Tabelle “File System Maximums”)

1. Wie groß kann das Dateisystem maximal werden? Hinweis: Multiplizieren Sie die Blockgröße (vgl. Lösung der Aufgabe 2.2.3) mit der maximalen Anzahl der Blöcke. Beachten Sie die Wortgröße!
2. Wie viele Inodes kann es maximal geben? Hinweis: Jeder Inode hat (unter x86) eine 4-Byte-Inode-Nummer und keine zwei Inodes dürfen dieselbe Inode-Nummer haben.
3. Wie groß kann eine Datei, die in eine einzige Block-Gruppe passt, maximal sein? Hinweis: Die Ausgabe von `dumpe2fs` liefert einen Eintrag, der auf die Menge der Datenblöcke innerhalb einer Blockgruppe Aufschluss gibt. Nutzen Sie diesen Wert und erinnern Sie sich an die Blockgröße.

2.4 Einlesen des Superblocks

Schreiben Sie ein C-Programm mit dem sie den Superblock einlesen und einige Informationen auf der Kommandozeile ausgeben.

- Anzahl der Inodes
- Insgesamte Anzahl der Blöcke
- Erster Datenblock
- Anzahl freier Inodes
- Anzahl freier Datenblöcke
- Magic Number (Ausgabe in Hex)

¹Tatsächlich werden Sie jedoch auf einem 64-Bit-System arbeiten

- 128-bit UUID des Dateisystems (Ausgabe in Hex)
- Betriebssystem des Erstellers

```

/* Blaupause fuer die Loesung */
#include <ext2fs/ext2fs.h>
#include <ext2fs/ext2_fs.h>
#include <stdio.h>
#include <stdlib.h>
#include <et/com_err.h>
#include <stdint.h>

static const char DEV[] = "/dev/loop0p1";
static const char OS_L[] = "LINUX";
static const char OS_O[] = "OTHER";

int main(int argc, char **argv){
    errcode_t err;
    ext2_filsys fs;
    struct ext2_super_block *super;

    /* Oeffnen des Dateisystemhandles "fs" mithilfe der Funktion
       ext2fs_open.
       Hinweis: Verwenden Sie Kapitel 4 als Referenz und nutzen Sie
       EXT2_FLAG_RW, 0, 0, unix_io_manager als Argument 2 - 5 fuer die
       Funktion!
       Argument 1 und 6 muessen Sie selbst herausfinden. */
    ???

    if (err){
        if (err == 13) {
            printf("[ext2fs_open]_Insufficient_permissions\n");
        } else {
            printf("[ext2fs_open]_Error_number:_%ld\n", err);
        }
        return EXIT_FAILURE;
    }

    super = fs->super;

    /* Hinweis: Hier printf-Statements der folgenden Member von super:
       super->s_inodes_count (integer)
       super->s_blocks_count (integer)
       super->s_first_data_block (integer)
       super->s_free_inodes_count (integer)
       super->s_free_blocks_count (integer)
       super->s_magic (%02X)
       super->s_uuid[] (verwenden Sie *((uint32_t *) super->s_uuid) +
       <Index>)
       super->s_creator_os
       (integer. Geben Sie "Linux" aus,
       wenn Wert gleich 0, sonst "Other") */
    ???

    /* Schliessen des Dateisystem-Handles mittels ext2fs_close(fs).
       Nutzen Sie die in Kapitel 4 dokumentierte Funktion
       ext2fs_close(ext2_filsys FS)*/
    ???

    if (err){
        printf("[ext2fs_close]_Error_number:_%ld\n", err);
    }
}

```

```

        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

```

2.5 Anlegen von Testdaten

1. Erstellen Sie im neuen Dateisystem den angegebenen Verzeichnisbaum. Hierfür steht Ihnen ein Bash-Skript zur Verfügung (vgl. `scripts/file_tree.sh`). Der Aufruf lautet `sudo ./file_tree.sh`. Das Skript wechselt automatisch ins richtige Verzeichnis und wieder zurück. Der folgende Baum wird generiert:

```

/+--> t1
|   +--> dirA
|       +--> foo.txt -- Inhalt: "FOO.TXT"
|   +--> a.txt -- Inhalt "A"
|   +--> b.txt -- Inhalt "B"
|   +--> c.txt -- 2048 mal "abc "
+--> t2
|   +--> dirB
|   +--> dirC
+--> foo.txt -- Inhalt "FOO"

```

2.6 Debugfs

In dieser Aufgabe machen Sie sich mit dem Dateisystemdebugger `debugfs` vertraut.

1. Starten Sie `debugfs` (`sudo debugfs /dev/loop0p1`).
2. Welcher Inode stellt das Verzeichnis “t1” dar, welcher “t2”? (Hinweis `ls -l2`)
3. Welcher Inode stellt die Verzeichniswurzel dar und auf welchem Block (i.S.v. Ext4) liegt dieser? (Hinweis `imap <inode nummer> ODER <pfad>`)
4. Wechseln Sie mit `cd` in das Verzeichnis “t1” und lassen Sie sich alle Inodes anzeigen. Benutzen Sie das “cat”- Kommando, um sich den Inhalt der Dateien `a.txt` und `b.txt` ausgeben zu lassen.
5. Wo liegt der Block des Inodes für `a.txt`?
6. Lassen Sie sich mittels `stat <inode nummer>` den Inodestatus für die Datei `a.txt` ausgeben. Auf welchem Block liegen die Daten?
7. Geben Sie den Datenblock mittels `block_dump` aus.
8. Wie viele Blocks benötigt die Datei `c.txt`? (Hinweis: `blocks <inode nummer>`)
9. Geben Sie auch noch einen der ausgegebenen Blöcke auf der Standardausgabe aus (`block_dump`).

² Es handelt sich hierbei um ein Kommando von `debugfs`, ist also insbesondere nicht mit dem `ls`-Kommando zu verwechseln.

2.7 Inodes

1. Lassen Sie sich den Inode der Datei *a.txt* mithilfe der Kommandos `dd` und `hexdump` anzeigen. Hinweis: Suchen Sie zunächst die Nummer des Inodes. Nun können Sie die Blocknummer des Inodes und dessen Offset zu diesem Block ermitteln. Erinnern Sie sich auch daran, dass ein Inode typischerweise 128 Byte groß sein wird!
2. Schreiben Sie ein C-Programm mit dem Sie sich die folgenden Daten des Inodes der Datei *c.txt* ausgeben lassen. Nutzen Sie dazu die angegebene Vorlage und Kapitel 4.3 und achten Sie auf die Fragezeichen im Code.

- User-ID
- Modifikationsdatum (nur als Integer ausgeben)
- Zugriffsdatum (nur als Integer ausgeben)
- Berechtigungen (Dateimodus)
- Anzahl der Blöcke in Bytes

```
#include <ext2fs/ext2fs.h>
#include <ext2fs/ext2_fs.h>
#include <stdio.h>
#include <stdlib.h>
#include <et/com_err.h>
#include <stdint.h>

static const char DEV[] = "/dev/loop0p1";
static const int INODE = ???;

int main(int argc, char *argv[]){
    errcode_t err;
    struct struct_ext2_filsys *fs;
    struct ext2_inode inode;

    err = ext2fs_open(DEV, EXT2_FLAG_RW, 0, 0, unix_io_manager, &fs);
    if (err){
        if (err == 13) {
            printf("[ext2fs_open]_Insufficient_permissions\n");
        } else {
            printf("[ext2fs_open]_Error_number:_%ld\n", err);
        }
        return EXIT_FAILURE;
    }

    ???
    err = ext2fs_read_inode(fs, INODE, &inode);
    if (err){
        printf("[ext2fs_read_inode]_Could_not_read_inode_%d\n", INODE);
        return EXIT_FAILURE;
    }

    /* Hier printf-Statements einfüegen.
       inode.i_uid -> User-id (integer)
       inode.i_mtime -> Modifikationsdatum (integer)
       inode.i_atime -> Zugriffsdatum (integer)
       inode.i_mode -> Berechtigungen (integer, oktal!)
       inode.i_blocks -> Blockanzahl (integer) */
    ???

    err = ext2fs_close(fs);
```

```

    if (err){
        printf("[ext2fs_close]_Error_number:_%ld\n", err);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

```

3 Blockgruppen Layout von Ext4

Die folgende Tabelle beschreibt den grundsätzlichen Aufbau einer ext4-Blockgruppe. Die Menge der Blöcke, die in einer Blockgruppe enthalten sind, lässt sich leicht mithilfe des `dumpe2fs`-Kommandos ermitteln (vgl. auch 2.3).

Daten	Größe
Füllbytes vor der Gruppe 0	1024 Bytes
Super Block	1 Block (1KiB, 2KiB, 4KiB, ..., 64KiB)
Group-Descriptors	n Blöcke
Reservierte Group-Descriptors	k Blöcke. Werden für's nachträgliche Vergrößern benötigt
Datenblock Bitmap	1 Block
Inode Bitmap	1 Block
Inode Tabelle	j Blöcke (Im Test-Dateisystem 8 1KiB-Blöcke)
Datenblöcke	Der Rest

Tabelle 1: Blockgruppen Layouttabelle

4 Ext4-API-Referenz

Im Folgenden einige Auszüge aus der offiziellen API-Referenz für das Ext4-Dateisystem.

4.1 Öffnen eines ext4-Dateisystems

Most libext2fs functions take a filesystem handle of type 'ext2_filsys'. A filesystem handle is created either by opening an existing function using 'ext2fs_open', or by initializing a new filesystem using 'ext2fs_initialize'.

```
-- Function: errcode_t ext2fs_open (const char *NAME, int FLAGS, int
SUPERBLOCK, int BLOCK_SIZE, io_manager MANAGER, ext2_filsys
*RET_FS)
```

Opens a filesystem named NAME, using the the io_manager MANAGER to define the input/output routines needed to read and write the filesystem. In the case of the 'unix_io' io_manager, NAME is interpreted as the Unix filename of the filesystem image. This is often a device file, such as '/dev/hda1'.

The SUPERBLOCK parameter specifies the block number of the

superblock which should be used when opening the filesystem. If SUPERBLOCK is zero, 'ext2fs_open' will use the primary superblock located at offset 1024 bytes from the start of the filesystem image.

The BLOCK_SIZE parameter specifies the block size used by the filesystem. Normally this is determined automatically from the filesystem superblock. If BLOCK_SIZE is non-zero, it must match the block size found in the superblock, or the error 'EXT2_ET_UNEXPECTED_BLOCK_SIZE' will be returned. The BLOCK_SIZE parameter is also used to help fund the superblock when SUPERBLOCK is non-zero.

The FLAGS argument contains a bitmask of flags which control how the filesystem open should be handled.

'EXT2_FLAG_RW'

Open the filesystem for reading and writing. Without this flag, the filesystem is opened for reading only.

'EXT2_FLAG_FORCE'

Open the filesystem regardless of the feature sets listed in the superblock.

4.2 Schließen eines Ext4-Dateisystems

-- Function: `errcode_t ext2fs_flush (ext2_filsys FS)`

Write any changes to the high-level filesystem data structures in the FS filesystem. The following data structures will be written out:

- * The filesystem superblock
- * The filesystem group descriptors
- * The filesystem bitmaps, if read in via 'ext2fs_read_bitmaps'.

-- Function: `void ext2fs_free (ext2_filsys FS)`

Close the io_manager abstraction for FS and release all memory associated with the filesystem handle.

-- Function: `errcode_t ext2fs_close (ext2_filsys FS)`

Flush out any changes to the high-level filesystem data structures using 'ext2fs_flush' if the filesystem is marked dirty; then close and free the filesystem using 'ext2fs_free'.

4.3 Lesen eines Inodes

-- Function: `errcode_t ext2fs_read_inode (ext2_filsys FS, ext2_ino_t INO, struct ext2_inode *INODE)`

Read the inode number INO into INODE.


```
-- Function: errcode_t ext2fs_write_inode (ext2_filsys FS, ext2_ino_t  
INO, struct ext2_inode *INODE)  
Write INODE to inode INO.
```