Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 11: Dateisysteme

zum 14. Januar 2019

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 20.01.2019 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt11.tgz zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `int chdir(const char *path);`
- `char *strerror(int errnum);`
- `char *getcwd(char *buf, size_t size);`

2 Tutoraufgaben

2.1 Dateisystem in UNIX

2.1.1 Maximale Dateigröße

Der Hintergrundspeicher ist in Blöcke aufgeteilt. Nehmen Sie im Folgenden an, ein Block habe eine Größe von 1 KiB und die Adresse eines Blocks benötige 32 Bit. Welche Länge (in Byte) kann eine Datei maximal haben, wenn im I-Node der Datei neben den single-, double- und triple-indirect-Verweisen noch 10 Datenblöcke direkt referenziert werden? Begründen Sie Ihre Antwort.

2.2 EXT im Detail

a) Im ext Dateisystem werden alle Objekte mittels I-Nodes verwaltet. Welche Informationen müssen innerhalb eines I-Nodes gespeichert werden?

b) Warum ist der Name einer Datei nicht in seiner I-Node gespeichert?

c) Was ist bei dem Dateisystem ext unter einem Link zu verstehen?

d) Was ist der Unterschied zwischen einem Hardlink und einen Softlink? Wie sind die beiden Typen implementiert?

e) Bei Linux wird das Dateisystem immer als / zur Verfügung gestellt. Wie können dabei weitere Dateisysteme eingebunden werden?

2.3 Vergleich: I-Nodes und FAT

Auf einer 16 GiB-Festplatte werden Dateien unter UNIX mittels I-Nodes gespeichert. Ein I-Node verfüge im Folgenden über 10 direkte Referenzen auf Datenblöcke, einen einfach indirekten Verweis und einen zweifach indirekten Verweis. Die Blockgröße beträgt 1024 Bytes und die Adresslänge 32 Bit.

a) Wie viele Blöcke Verwaltungsaufwand benötigt eine 250 KiB-Datei?

b) Welchen Umfang würde eine FAT (File Allocation Table) für die obige Festplatte haben? Verwenden Sie die kleinstmögliche Anzahl von Bits für die Blocknummern.

c) Welches der beiden Verfahren (I-Nodes, FAT) ist bei der Verwaltung von wenigen kleinen Dateien im Vorteil? Warum? Begründen Sie Ihre Antwort (denken Sie an den Hauptspeicherbedarf).

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

2.4 Das FAT16-Dateisystem

Machen Sie sich in dieser Aufgabe mit dem Aufbau des Dateisystems FAT16 vertraut.

a) Schildern sie kurz den Aufbau einer FAT16 Partition.

b) Was ist bei Festplatten unter den Begriffen **Sektor** und **Cluster** zu verstehen.

c) Beschreiben Sie kurz die einzelnen Elemente der `boot_sector` Datenstruktur von Fat16.

d) Beschreiben Sie kurz, wie Dateien innerhalb von Fat16 dargestellt werden.

2.5 Journaling in Dateisystemen (Tutoraufgabe)

a) Welche Operationen müssen beim Löschen einer Datei durchgeführt werden?

b) Was verbirgt sich unter dem Konzept des Journaling? Gegen welches Problem hilft es?

c) Welche Operationen müssen auf dem Dateisystem durchgeführt werden, wenn das Dateisystem kein Journaling unterstützt und das Löschen einer Datei nicht erfolgreich beendet werden konnte (durch Stromausfall)?

d) Welchen Zweck hat der Ordner `lost+found` im root Verzeichniss eines ext Dateisystems?

e) Wie wird das Schreiben in das Dateisystem beim Vorhandensein von Journaling durchgeführt?

2.6 Dateisystem (Ext4): Dateizugriff im Detail

In der letzten Woche haben wir uns Verzeichnisinhalte unter Nutzung der entsprechenden System-Calls angesehen. In dieser Aufgabe wollen wir die interne Funktionsweise des Dateisystems – vereinfacht – nachvollziehen.

Ihre GBS-VM verwendet das Dateisystem ext4 (ein Journaling-Dateisystem), das an verschiedenen Stellen von ext2/ext3 abweicht. Darum werden wir an einigen Stellen leicht von den Vorlesungsfolien abweichen müssen, vor allem, wenn es um das Auffinden von Datenblöcken zu einer I-Node geht.

Wir wollen ein Programm “fs” schreiben, das als Dateinamen das Gerät (konkret: die Festplattenpartition) übergeben bekommt und dann für diese Partition eine Datei findet und deren Inhalt ausgibt. Die Partition Ihres Linux-Systems heißt `/dev/sda1`. Sie können Sie nur als Superuser öffnen und sollten das auch nur “Read-only” tun.

Hierzu stellen wir zunächst sechs Funktionen zur Verfügung, damit Sie sich nicht durch die Details des Ext4-Dateisystems lesen müssen. Damit können Sie:

a) read_block() – den angegebenen Block lesen

b) extract_superblock() – Details des Superblocks extrahieren und teils in globalen Variablen ablegen

c) read_inode() – einen I-Node lesen

d) get_data_blocks() – die ersten k zu einer Datei gehörenden Blöcke aus einer I-Node extrahieren

e) print_inode() – einige Details zu einer I-Node ausgeben

f) search_dir() – alle Dateien eines Verzeichnisses ausgeben

Details zu Ext4:

`https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`

```
int read_block (int fd, char *buffer, int block_no) {
    int                 len;
    unsigned long       offset = (unsigned long) block_no * (unsigned long) block_size;

    /* position read pointer at the beginning of the block, blocks count from zero */
    if (lseek (fd, offset, SEEK_SET) == -1) {
        perror ("lseek failed");
        return -1;
    }
    /* read the block contents */
    if ((len = read (fd, buffer, block_size)) == -1) {
        perror ("Cannot read root directory inode");
        return -1;
    }
    return len;
}

void extract_superblock_info (char *buffer) {
    char    *s;

    s = buffer + SUPERBLOCK_OFFSET;
    printf ("#i-nodes =\t%d\n", *((int *) (s + N_INODES)));
    printf ("#blocks =\t%d\n", *((int *) (s + N_BLOCKS)));
    printf ("#free inodes =\t%d\n", *((int *) (s + FREE_BLOCKS)));
    printf ("#free blocks =\t%d\n", *((int *) (s + FREE_BLOCKS)));
    block_size = 1 << (10 + *((int *) (s + LOG_BLOCKSIZE)));
    printf ("#block size =\t%d\n", block_size);
    printf ("#blocks/group =\t%d\n", *((int *) (s + BLOCKS_PER_GROUP)));
    inodes_per_group =  *((int *) (s + INODES_PER_GROUP));
    printf ("#inodes/group =\t%d\n", inodes_per_group);
    printf ("first inode # =\t%d\n", *((int *) (s + FIRST_INODE)));
    inode_size = *((short *) (s + INODE_SIZE));
    printf ("inode size =\t%d\n", inode_size);
    sb_extents = *((int *) (s + INCOMPAT)) & 0x40;
    printf ("volume label =\t%s\n", s + VOLUME_LABEL);
    printf ("prealloc file =\t%d\n", *(s + PREALLOC_FILE));
    printf ("reserved GDT =\t%d\n", *((short *) (s + RESERVED_GDT)));
}

char *read_inode (int fd, char *buffer, int inode) {
    int    block_no;
    int    offset;
    int    len;

    block_no = inode_table + (inode - 1) * inode_size / block_size;
    offset = ((inode - 1) * inode_size) % block_size;

    if ((len = read_block (fd, buffer, block_no)) == -1) {
        perror ("Cannot read inode");
        return NULL;
    }
    return buffer + offset;
}

int get_data_blocks (char *buffer, int *blocks, int *nblocks, int inode) {
    char        *s = buffer;
    int         file_size, extents, i, j, k;
    int         entries, n;

    file_size = *((int *) (s + INODE_FILE_SIZE));
    extents = *((int *) (s + INODE_FLAGS)) & 0x80000;
```

```
    if (!extents) {
        /* old structure using just direct and indirect blocks */
        for (i = 0; i < file_size / block_size && i < *nblocks; i++) {
            blocks [i] = *((int *) (s + INODE_FILE_BLOCKS + 4*i));
        }
        *nblocks = i;
    } else {
        /* new structure using extents -> lazy and just read the first block for now */
        s += INODE_FILE_BLOCKS;
        printf ("\textent magic number = %x\n", *((short *) (s + EXTENT_MAGIC)) & 0x00ffff);
        entries = *((short *) (s + EXTENT_ENTRIES));
        printf ("\textent # entries = %x\n", entries);
        printf ("\textent depth = %x\n", *((short *) (s + EXTENT_DEPTH)));
        s += 12; /* extent tree header size */
        /* leaf nodes */
        i = 0;
        for (j = 0; j < entries && i < *nblocks; j++) {
            printf ("\textent start block = %d\n", *((int *) (s)));
            n = *((short *) (s + 4));
            printf ("\textent # blocks = %d\n", n);
            printf ("\tstart block number = %d\n", *((int *) (s + 8)));
            /* add n consecutive blocks starting at the indicated block number */
            for (k = 0; k < n && i < *nblocks; k++)
                blocks [i++] = *((int *) (s + 8)) + k;
            s += 12; /* next extent block */
        }
        *nblocks = i;
    }
    printf ("Data blocks of inode %d:", inode);
    for (i = 0; i < *nblocks; i++) {
        printf ("\t%s%d%s", i > 11 ? "[" : "", blocks [i], i > 11 ? "]" : "");
    }
    printf ("\n");
    return file_size;
}

void print_inode (char *s, int inode) {
    int         extents, file_size;

    printf ("i-node %d\n", inode);
    extents = *((int *) (s + INODE_FLAGS)) & 0x80000;
    printf ("\tinode flags = \%x\n", *((int *) (s + INODE_FLAGS)));
    printf ("\tmode = %o\n", *((short *) s) & 00777);
    printf ("\ttype = %x\n", *((short *) s) & 0x0f000);
    file_size = *((int *) (s + INODE_FILE_SIZE));
    printf ("\tfile size = %d\n", file_size);
    printf ("\tlinks = %d\n", *((short *) (s + INODE_LINK_COUNT)));
    printf ("\textent = %s\n", extents ? "YES" : "NO");
}

int search_dir (int fd, int *blocks, int nblocks, char *target) {
    int         inode = 0, len, i;
    char        buffer [BLOCK_SIZE], *s;
    char        fn [256];

    for (i = 0; i < nblocks; i++) {
        if ((len = read_block (fd, buffer, blocks [i])) == -1) {
            close (fd);
            return -1;
        }
        /* print directory contents */
        s = buffer;
        while (s < buffer + len) {
            strncpy (fn, s + DIR_FILE_NAME, *((unsigned char *) (s + DIR_NAME_LEN)));
            fn [*((unsigned char *) (s + DIR_NAME_LEN))] = '\0';
            printf ("\t%d\t%x\t%s\n", *((int *) s), *(s + DIR_FILE_TYPE), fn);
            /* we want to look at /etc, so check when we find this directory */
```

```
            if (!strcmp (fn, target))
                inode = *((int *) s);
            s += *((short *) (s + DIR_REC_LEN));
        }
    }
    return inode;
}
```

Sie können außerdem eine Reihe von Definitionen annehmen:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

extern int hexdump (FILE *, char *, int);

#define BLOCK_SIZE              4096
#define SUPERBLOCK              0
#define SUPERBLOCK_OFFSET       1024
#define ROOT_DIR_INODE          2

/* super block offsets */
#define N_INODES                0x00
#define N_BLOCKS                0x04
#define FREE_INODES             0x10
#define FREE_BLOCKS             0x0c
#define FIRST_DATA_BLOCK        0x14
#define LOG_BLOCKSIZE           0x18
#define BLOCKS_PER_GROUP        0x20
#define INODES_PER_GROUP        0x28
#define FIRST_INODE             0x54
#define INODE_SIZE              0x58
#define INCOMPAT                0x60
#define COMPAT                  0x64
#define VOLUME_LABEL            0x78
#define PREALLOC_FILE           0xcc
#define RESERVED_GDT            0xce

/* block group offsets */
#define INODE_TABLE             0x08

/* inode offsets */
#define INODE_FILE_SIZE         0x04
#define INODE_FLAGS             0x20
#define INODE_LINK_COUNT        0x1a
#define INODE_N_BLOCKS          0x1c
#define INODE_FILE_BLOCKS       0x28

/* file blocks in ext4 can be managed via extent trees */
#define EXTENT_MAGIC            0x00
#define EXTENT_ENTRIES          0x02
#define EXTENT_DEPTH            0x06

/* directory entries */
#define DIR_INODE               0x00
#define DIR_REC_LEN             0x04
#define DIR_NAME_LEN            0x06
#define DIR_FILE_TYPE           0x07
#define DIR_FILE_NAME           0x08

int     block_size = 4096;
int     inode_size, inodes_per_group, inode_table, sb_extents;
```

Schreiben Sie jetzt ein Programm, das die Datei /usr/include/ctype.h findet und den Inhalt des ersten Datenblocks dieser Datei als Hexdump ausgibt. In einem Dateisystem müssen Sie hierzu zunächst den

Superblock lesen und dann die erste Blockgruppe bei Ext4, um die I-Node-Tabelle zu finden. Dann beginnen Sie mit dem Root-Verzeichnis (I-Node 2) und arbeiten sich von dort aus I-Node für I-Node und Datenblock für Datenblock durch die Verzeichnisse, bis Sie die entsprechende Datei gefunden haben.

Achtung: Die obigen Funktionen sind sehr rudimentär implementiert und berücksichtigen beispielsweise nicht mehrere Blockgruppen und unterstützen auch keine großen Dateien. Die I-Nodes aller Verzeichnisse müssen sich deshalb in der ersten Blockgruppe befinden (hier konkret < 8192 sein).

Sehen Sie sich auch mal (beispielsweise bei obigem Link) an, welchen Daten in der Praxis alle im Superblock und in den I-Nodes gespeichert werden. Wenn Sie an der Interpretation weiteren Daten Interesse haben, können Sie das Programm entsprechend erweitern.

3 Hausaufgaben

3.1 Ergänzungen zur Shell

3.1.1 Verzeichnisse in der Shell (3 Punkte)

In einer typischen Unix-Shell befinden Sie sich zu jedem Zeitpunkt in einem aktuellen Arbeitsverzeichnis. Dieses wird (in der bash) auch in der Variable PWD reflektiert. Zum Wechseln des Verzeichnisses implementiert die Shell das bekannte Kommando `cd`.

Ergänzen Sie Ihre Shell um das interne Kommando `cd` und verwalten Sie das aktuelle Arbeitsverzeichnis intern. Bilden Sie es auch in der Variables PWD ab. Geben Sie als Prompt das jeweils aktuelle Verzeichnis aus:

```
/home/student $
```

Wenn das Kommando `cd` ohne Parameter aufgerufen wird, geben Sie das aktuelle Verzeichnis aus. Sehen Sie auch die notwendige Fehlerbehandlung vor:

```
/home/student $ cd /x
cd: /x: No such file or directory
/home/student $ cd x y
Usage: cd <dir>
```

Nutzen Sie zur Ausgabe der Fehlermeldung die Funktion `strerror()`, und geben Sie diese auf `stdout` aus.

Beispiel:

```
$ ./mysh
/tum/git/gbs/uebungsbetrieb/ws18/11/c-files $ ls
Makefile        list.c          list.h          mysh            mysh.c          parser.c
/tum/git/gbs/uebungsbetrieb/ws18/11/c-files $ cd ..
/tum/git/gbs/uebungsbetrieb/ws18/11 $ ls
Makefile                blatt11-content.tex~    blatt11.pdf             gbsuebung.cls
blatt11-content.tex     blatt11-loesung.pdf     c-files                 uebung.sty
/tum/git/gbs/uebungsbetrieb/ws18/11 $ cd cd code
Usage: cd <dir>
/tum/git/gbs/uebungsbetrieb/ws18/11 $ cd code
cd: code: No such file or directory
/tum/git/gbs/uebungsbetrieb/ws18/11 $ cd
/tum/git/gbs/uebungsbetrieb/ws18/11
/tum/git/gbs/uebungsbetrieb/ws18/11 $
```

3.1.2 Wildcards in der Shell (7 Punkte)

In einer Shell können Sie auf der Kommandozeile Wildcards eingeben wie im einfachen Fall beispielsweise:

```
$ ls *.c
list.c  mem.c  memory.c  t.c  t2.c
```

Diese Wildcards werden **nicht** durch das Programm `ls` interpretiert, sondern durch die Shell selbst. Dies können Sie einfach sehen, indem Sie in der Shell eingebaute Kommando `echo` verwenden:

```
$ echo *.c
list.c mem.c memory.c t2.c t.c
$ echo ../*.pdf *.h
../blatt09-loesung.pdf ../blatt09.pdf list.h memory.h
```

Konkret bedeutet das, dass die Shell das entsprechende Verzeichnis bzw. die entsprechenden Verzeichnisse öffnet und nach passenden Einträgen durchsucht, um anschließend jeden Ausdruck mit Wildcard durch die jeweilige Liste zu ersetzen. Im obigen Fall wird also "`../*.pdf`" expandiert zu "`../blatt09-loesung.pdf ../blatt09.pdf`" und "`*.h`" zu "`list.h memory.h`".

Achtung: Es muss nicht das aktuelle Verzeichnis sein, sondern es kann sich auch um einen absoluten Pfad oder einen Pfad relative zum aktuellen Verzeichnis handeln. Achten Sie auch darauf, den jeweiligen (relativen oder absoluten) Pfad allen hierzu gehörenden Dateien voranzustellen.

Unterstützen Sie folgende Formen von Wildcards:

- Wildcard anstelle des Dateinamens: `*` oder `*/Makefile`
- Wildcard am Anfang des Dateinamens: `*.c` oder `code/*.c` oder `aufgabe*/Makefile`
- Wildcard am Ende des Dateinamens: `blatt*` oder `code/list.*`
- Genau ein Wildcard in der Mitte des Dateinamens: `blatt*.tex` oder `code/mem*.c`

Das heißt auch, dass Sie komplexe Ausdrücke wie beispielsweise mehrere Wildcards **nicht** unterstützen müssen (also etwa `*/*.c` oder `a*b*c*d.txt`.

Zum Testen können Sie das Kommando `echo` verwenden, das alle übergebenen Parameter durch einzelne Leerzeichen getrennt ausgibt.

Hinweis: Sie können die Verzeinisse selbst komplett lesen und manuell unter Verwendung verschiedener String-Funktionen (etwa `strnstr()` oder `strncmp()` parsieren und die passenden Einträge extrahieren. Oder aber Sie verwenden die Funktion `scandir()`, um die passenden Dateinamen aufzufinden.

Hinweis: Die typische Shell unterdrückt bei der Ausgabe alle Dateien, die mit einem Punkt (".") beginnen. In Ihrer Lösung sollen Sie die beiden Einträge "." und ".." unterdrücken, d.h., überspringen Sie diese beiden Einträge im Verzeichnis. Für alle anderen Dateien, die mit "." beginnen, braucht dieses Verhalten **nicht** nachgebildet werden.

Beispiele:

```
$ ./mysh
/home/student/gbs/uebungsbetrieb/ws18/11/c-files $ ls
Makefile  list.c  list.h  mysh  mysh.c  parser.c  wildcard.c  wildcard.h
/home/student/gbs/uebungsbetrieb/ws18/11/c-files $ cd ..
/home/student/gbs/uebungsbetrieb/ws18/11 $ echo *
blatt11-content.tex uebung.sty gbsuebung.cls Makefile . .. c-files
```

```
/home/student/gbs/uebungsbetrieb/ws18/11 $ cd ..
/home/student/gbs/uebungsbetrieb/ws18 $ cd 11
/home/student/gbs/uebungsbetrieb/ws18/11 $ ls ../*/Makefile
../01/Makefile  ../02/Makefile  ../03/Makefile  ../04/Makefile  ../05/Makefile  ../06/Makefile
../07/Makefile  ../08/Makefile  ../09/Makefile  ../10/Makefile  ../11/Makefile
/home/student/gbs/uebungsbetrieb/ws18/11 $ ls ../1*/Makefile
../10/Makefile  ../11/Makefile
/home/student/gbs/uebungsbetrieb/ws18/11 $ ls ../*1/Makefile
../01/Makefile  ../11/Makefile
/home/student/gbs/uebungsbetrieb/ws18/11 $ cd 11
cd: 11: No such file or directory
/home/student/gbs/uebungsbetrieb/ws18/11 $ ls M*e
Makefile
/home/student/gbs/uebungsbetrieb/ws18/11 $ ls c-files/*.h
c-files/list.h  c-files/wildcard.h
/home/student/gbs/uebungsbetrieb/ws18/11 $ echo c-files/*.c
c-files/wildcard.c c-files/list.c c-files/parser.c c-files/mysh.c
/home/student/gbs/uebungsbetrieb/ws18/11 $ echo c-files/*
c-files/wildcard.c c-files/Makefile c-files/list.c c-files/mysh c-files/. c-files/list.h c-files/..
c-files/parser.c c-files/wildcard.h c-files/mysh.c
```

3.1.3 Code

Beide Teilaufgaben werden durch eine Abgabe eingereicht:

Syntax: `mysh`

Quellen: mysh.c list.c list.h parser.c wildcard.c wildcard.h

Executable: mysh