Technische Universität Berlin Fachgebiet Distributed and Operating Systems Sommersemester 2025	**Praxisaufgabe 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**20.07.2025 23:59 Uhr**

In dieser Aufgabe soll ein virtuelles Dateisystem implementiert werden. In diesem sollen in einer Datenstruktur, die den in der Vorlesung vorgestellten INodes ähneln, Daten geordnet gespeichert werden können. Die Daten sollen gelesen und geschrieben werden können. Alle diese Daten werden dann in einer gemeinsamen Datei gespeichert, welche auf eurem Rechner liegt. Die hier genutzten INodes unterscheiden sich allerdings in den Details von den in der Vorlesung vorgestellten INode, deshalb ist es wichtig, die Hinweise auf diesem Aufgabenblatt gründlich zu lesen.

Eine Filsystem-Datei soll an der Dateiendung `.fs` erkennbar sein (muss aber nicht) und besteht aus den Folgenden Blöcken:

- Superblock, welcher
  - die Gesamtanzahl der Blöcke
  - die Anzahl der freien Blöcke

  beinhaltet.
- Freispeicher, als Array der freien Blöcke
- INodes
- Datenblöcke, welche die Daten der Dateien beinhalten

Den genauen Aufbau der einzelnen Bestandteile könnt ihr den struct-Definitionen in der `filesystem.h` oder den nachfolgenden Grafiken entnehmen.

Das Programm wird mit einem der beiden folgenden Befehle gestartet:

```
./build/ha2 -c <FILESYSTEM_NAME> <FS_SIZE>
./build/ha2 -l <FILESYSTEM_NAME>
```

Das erste Kommando erzeugt ein neues Filesystem (eine Datei mit der Endung `.fs`) in welcher alle weiteren Operationen ausgeführt werden. Das zweite Kommando öffnet ein bereits bestehendes Dateisystem. `FILESYSTEM_NAME` ist dabei der Name der Datei und `FS_SIZE` ist die Größe des Dateisystems, d.h. die Anzahl der 1024-Byte Blöcke und der INodes.

Es öffnet sich eine shell, wie ihr sie bereits kennt aus der ersten Übungsaufgabe in der ihr die implementierten Kommandos eingeben könnt.

Das Dateisystem soll die Operationen in Tabelle **??** durchführen können.

Vorgegeben ist die komplette shell inklusive einlesen der Benutzereingaben, sowie die Kommandozeilenargumente, wie das einlesen der Dateisystem-Datei.

Um das Programm wieder zu beenden könnt ihr `quit` oder `exit` eingeben.

Macht euch vor der Bearbeitung vertraut mit der vorgegebenen Struktur. Eure Aufgaben bestehen in der Implementation einiger Funktionen der `operations.c` Datei, lest euch also

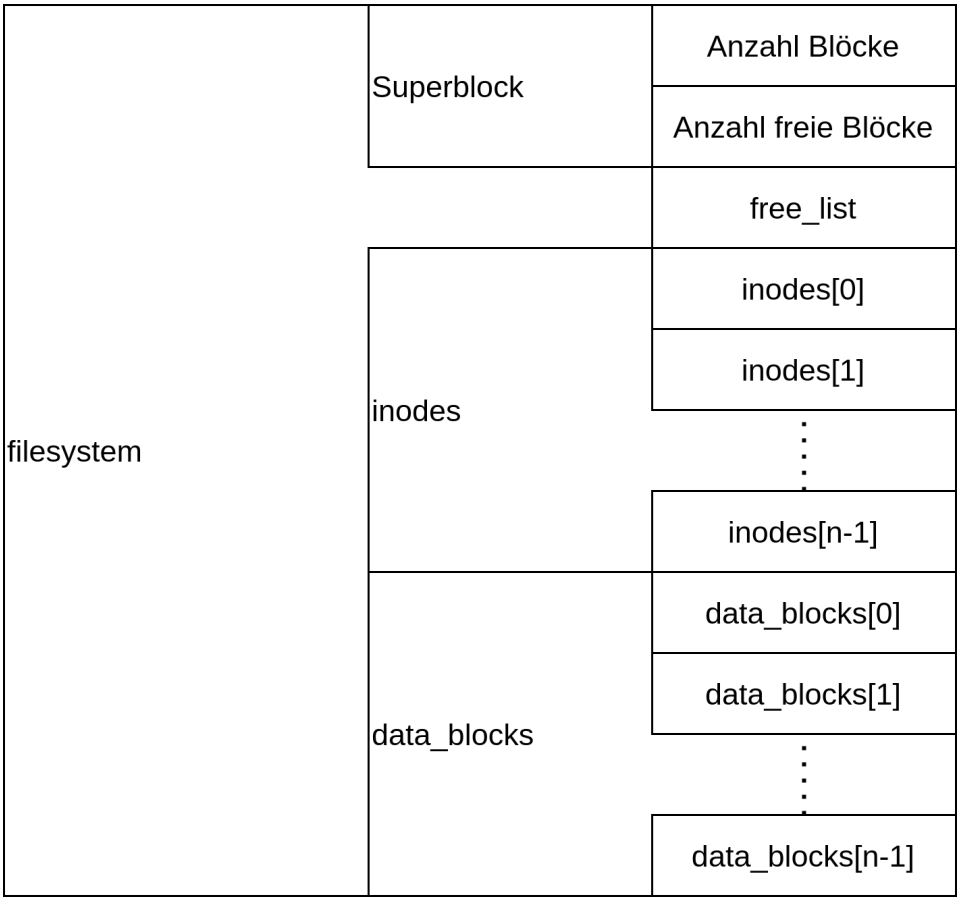

Abbildung 1: Übersicht über den Aufbau des Dateisystems

Kommando	Beschreibung
`mkdir <Pfad>`	Ein neues Verzeichnis anlegen
`mkfile <Pfad>`	Eine neue Datei anlegen
`cp <Pfad> <Pfad/Name>`	Kopiere ein Objekt zum angegebenen Pfad mit dem gegebenen Namen
`list <Absoluter Pfad>`	Den Inhalt eines Verzeichnisses anzeigen
`writef <Pfad> <Text>`	Text in eine Datei schreiben
`readf <Pfad>`	Text aus einer Datei auslesen
`rm <Pfad>`	Eine Datei, oder ein Verzeichnis löschen
`export <interner Pfad> <externer Pfad>`	Eine Datei exportieren
`import <externer Pfad> <interner Pfad>`	Eine Datei importieren
`dump`	Dateisystem auf Festplatte sichern

Tabelle 1: Kommandos, die das Dateisystem verstehen soll, und ihre Beschreibung. Alle Pfade sind absolut aus dem Ursprung des Dateisystems anzugeben. „Interner Pfad“ bezeichnet den Pfad im Dateisystem, und „externer Pfad“ den, auf eurem Rechner.
Das Kommando `dump` ist bereits implementiert.

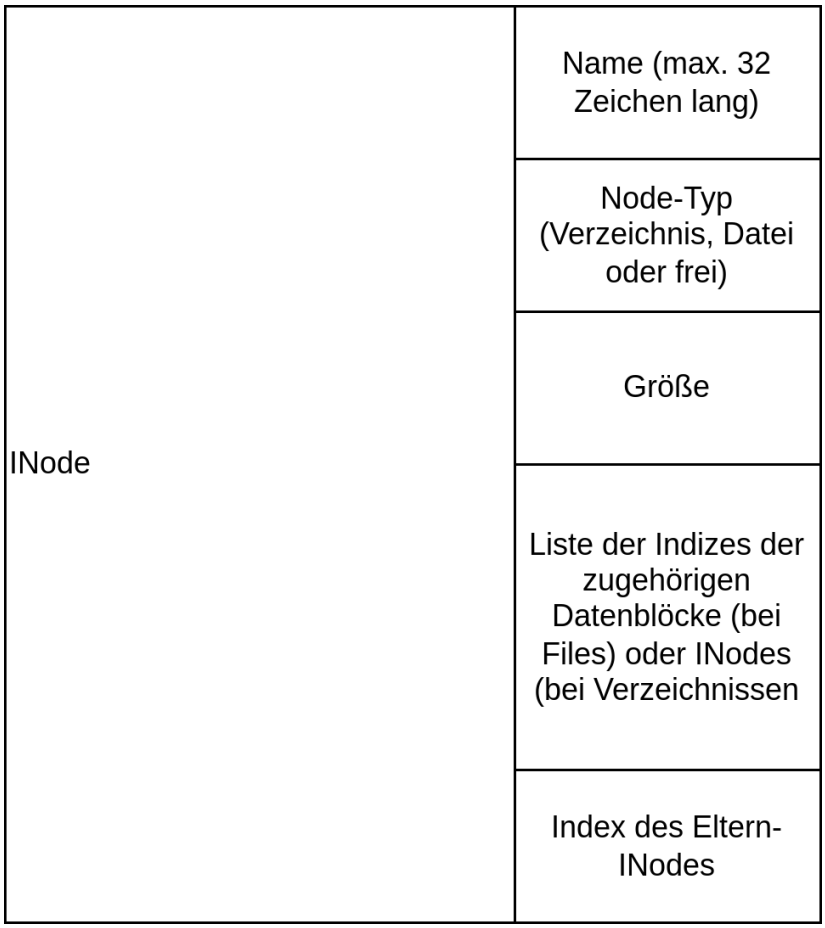

Abbildung 2: Aufbau einer INode. Wenn eine INode ein Verzeichnis repräsentiert, so stehen in dem Array „Array der Indizes“die Indizes der Kinder-INodes (weitere Verzeichnisse oder Dateien). Wenn die INode jedoch eine Datei repräsentiert, dann sind in diesem Array die Indizes der zugehörigen Blöcke zu finden.

<table>
<tr><td rowspan="6">filesystem</td><td rowspan="2">Superblock</td><td>Anzahl Blöcke = 2</td></tr>
<tr><td>Anzahl freie Blöcke = 1</td></tr>
<tr><td></td><td>free_list = [0,1]</td></tr>
<tr><td rowspan="2">inodes</td><td>inodes[0]:
name="/"
type=dir
direct_blocks[0] = 1
size = 0
parent = -1</td></tr>
<tr><td>inodes[1]:
name = "myFile"
type = file
direct_blocks[0] = 0
size = 5
parent = 0</td></tr>
<tr><td rowspan="2">data_blocks</td><td>data_blocks[0]:
size = 5
block = "Hello"</td></tr>
<tr><td>data_blocks[1]</td></tr>
</table>

Abbildung 3: Dateisystem mit 2 Blöcken mit Belegung. Es wurde die Datei `/myFile` erstellt und dieser die Daten `Hello` zugeordnet.

genau die Kommentare in der `operations.h` Datei durch. Sie geben wertvolle Hinweise zur Arbeitsweise der Funktionen. Das erste Argument, `file_system *fs` ist immer das Dateisystem auf dem gearbeitet werden soll.

Optional Erstellt euch weitere Testcases. Auch hier könnt ihr euch an den Vorgaben orientieren.

Aufgabe 1: `mkdir` 2 Punkte

Implementiert die Funktion `fs_mkdir`. Sie soll im Dateisystem einen neuen Ordner anlegen. Als Argument bekommt sie einen absoluten Pfad gegeben, an dem der Ordner erstellt werden soll. Um einen Ordner zu erstellen soll die kleinstmögliche freie INode verwendet werden. Setzt dazu den `n_type` auf `directory`, kopiert übergebenen Namen in das Namensfeld der INode und setzt den `parent` auf die INode-Nummer des übergeordneten Ordners. Im Parent-INode muss nun noch im `direct_blocks`-Array am kleinstmöglichen Eintrag die INode-Nummer des neuen Verzeichnisses eingetragen werden.

Aufgabe 2: `mkfile` 2 Punkte

Implementiert die Funktion `fs_mkfile`. Sie soll im Dateisystem eine neue Datei anlegen. Als Argument kriegt sie einen absoluten Pfad gegeben, an dem die Datei erstellt werden soll. Achtung: der Ordner in dem die Datei abgelegt wird muss vorher existieren. Das Erstellen einer Datei geschieht analog zum Erstellen eines Verzeichnisses, allerdings muss der INode-Type entsprechend auf `reg_file` gesetzt werden.

Aufgabe 3: `cp` 3 Punkte

Implementiert die Funktion `fs_cp`. Sie soll im Dateisystem eine Datei bzw. ein Verzeichnis kopieren. Als Argumente werden der Ursprungspfad und der Zeilpfad (letzter Teil des Zeilpfades ist der neue Name) übergeben.
Achtung: Verzeichnisse sollen rekursiv kopiert werden.
Hinweis: `cp /a /b/c`: Kopiere Verzeichnis `/a` in Verzeichnis `/b` und nenne es `c`.
(Für die Tests müssen `mkdir` und `mkfile` zuerst implementiert werden.)

Aufgabe 4: `list` 3 Punkte

Implementiert die Funktion `fs_list`. Als Argument bekommt sie einen absoluten Pfad gegeben. Es soll in dieser Funktion ein eine Liste aller in dem angegebenen Verzeichnis enthaltenen Dateien und Verzeichnisse als String erstellt und zurückgegeben werden. Diese sollen nach INode-Nummer sortiert sein. Die Namen dieser Elemente stehen jeweils in einer eigenen Zeile. Falls es sich um ein Verzeichnis handelt soll dem Namen `DIR` vorangestellt werden, bei einer Datei `FIL` und mit einem Leerzeichen vom Namen getrennt sein. Achtet darauf keine weiteren Zeichen in diesen String zu schreiben.

Ein Beispiel: Es wurden die Ordner „newDir“ (`inode[1]`) und „newerDir“ (`inode[3]`) und die Datei „myFile.txt“ (`inode[2]`) in „/“ erstellt. Dann führt `list /` zu dieser Ausgabe:

```
DIR newDir
FIL myFile.txt
DIR newerDir
```

Aufgabe 5: `writef` 4 Punkte

Implementiert die Funktion `fs_writef`. Als Argument bekommt sie einen absoluten Pfad, welcher die Datei angibt, die beschrieben werden soll, und den Text, der in diese Datei geschrieben werden soll, gegeben. Der Text soll nun in diese Datei geschrieben werden. Achtung: Die Datei muss bereits existieren. Wenn mehrfach in die gleiche Datei geschrieben wird, werden die weiteren Daten an die Datei angehängt, die vorhandenen Daten werden also nicht überschrieben. Folgendes soll beim Schreiben passieren, falls die Datei bisher leer ist: es soll der freie `data_block` mit dem geringsten Index gefunden werden, die Daten in das `block`-Feld dieses Blocks geschrieben werden, die `size` sowohl im `data_block` als auch in der INode auf die Länge der Daten gesetzt werden und die Ordnungszahl des benutzen blocks an der 0-ten Stelle des Array `inode->direct_blocks` gespeichert werden. Falls die zu schreibenden Daten größer sind als ein Block, dann müssen entsprechend mehrere freie Blöcke genutzt werden. Die benutzten Blöcke sollen in der Freiliste als "nicht frei"markiert werden, indem das Array `free_list` an der Stelle ihrer Ordnungszahlen auf 0 gesetzt werden.

Aufgabe 6: `readf` 4 Punkte

Implementiert die Funktion `fs_readf`. Diese soll den Inhalt einer Datei auslesen. Als Argument bekommt sie einen absoluten Pfad, welcher die Datei wählt, die ausgelesen werden soll. Außerdem bekommt sie den pointer zu einer Integervariable, in welche die Größe der Datei geschrieben werden soll. Die auszulesenden Daten sind jene in den `direct_blocks` der INodes, also die, die zum Beispiel mit `writef` in diese geschrieben wurden.

Aufgabe 7: `rm` 4 Punkte

Implementiert die Funktion `fs_rm`. Diese soll eine Datei, oder einen Ordner und seinen Inhalt rekursiv (auch den Inhalt der Ordner im Ordner usw.) löschen. Als Argument bekommt sie einen absoluten Pfad, welcher die Datei, oder den Ordner wählt, der gelöscht werden soll, enthält. Achtet darauf gelöschte INodes als frei zu markieren und auch eventuell benutzte Datenblöcke wieder für neue Daten nutzbar zu machen.

Aufgabe 8: `import` 4 Punkte

Implementiert die Funktion `fs_import`. Das Argument `int_path`, der interne Pfad, ist der Pfad zu einer Datei in dem Dateisystem, während das Argument `ext_path`, der externe Pfad, der Pfad zu einer (nicht unbedingt existierenden) Datei auf eurem Rechner ist. Die Funktion soll dann die Datei mit dem externen Pfad von Dateisystem eures Rechners lesen und den Inhalt in die angegebene interne Datei schreiben. Achtung: ähnlich wie bei `writef` muss auch hier die interne Datei schon existieren.

Aufgabe 9: `export` 4 Punkte

Implementiert die Funktion `fs_export`. Das Argument `int_path`, der interne Pfad, ist der Pfad zu einer Datei in dem Dateisystem, während das Argument `ext_path`, der externe Pfad, der Pfad zu einer (nicht unbedingt existierenden) Datei auf eurem Rechner ist. Die Funktion soll dann die Datei mit dem internen Pfad in das Dateisystem eures Rechners, an der Stelle des externen Pfades exportieren. Die externe Datei muss vorher noch nicht unbedingt existieren.

Hinweise:

- **Beschreibungen der Funktionen:** Nähere Beschreibungen der einzelnen Funktionen findet ihr auch in den .h-Dateien im Ordner lib.
- **Limitationen des Programms:** Die Länge von Dateinamen ist in diesem Programm auf 32 (inklusive Null-Byte) limitiert und die Anzahl der adressierbaren Blöcke pro Datei auf 12. Ihr müsst in eurem Programm nicht prüfen, ob diese Limits eingehalten werden. Geht also einfach immer davon aus, dass nie längeren Namen eingegeben werden, keine größeren Dateien erstellt werden und auch insgesamt nie mehr Speicher verwendet werden soll als in allen Blöcken insgesamt verfügbar.
- **Datei- und Verzeichnisnamen:** In einem Verzeichnis darf eine Datei und ein Unterverzeichnis nicht den gleichen Namen haben.
- **Vorgaben:** Bitte ändert bestehende Datenstrukturen, Funktionsnamen, Header-files, ... nicht. Eine Missachtung kann zu Punktabzug führen. Ändert nur die Datei `src/operations.c`. Gerne könnt ihr weitere Hilfsfunktionen oder Datenstrukturen innerhalb dieser Datei definieren und implementieren.
- **Makefile:** Bitte verwendet für diese Aufgabe das Makefile aus der Vorgabe. Führt hierzu im Hauptverzeichnis *make* aus. *make* kompiliert das Projekt mit *clang*. Unter Ubuntu können *make* und *clang* mit dem Befehl *sudo apt install make clang* installiert werden. Durch den Befehl *make clean* werden kompilierte Dateien gelöscht. Gerne könnt das Makefile um weitere Flags erweitern oder anderweitig anpassen. Euer Programm sollte aber mit dem vorgegebenen Makefile weiterhin kompilierbar bleiben.
- **Memoryleaks:** Überprüft abschließend euer Programm auf Memory leaks, um zu evaluieren ob der gesamte allozierte Speicher wieder freigegeben wurde. Hier empfiehlt sich das Kommandozeilenwerkzeug *valgrind*[1]. Memory leaks führen zu Punktabzug.
- **Tests:** Es stehen einige Tests zum Prüfen einzelner Funktionen im Verzeichnis `tests` zur Verfügung. Trotzdem wird das Erstellen eigener Tests sehr empfohlen, da die bereitgestellten **nicht** alles abdecken.
  Um die bereitgestellten Tests nutzen zu können muss python und das Modul pytest installiert sein. Falls diese Programme bei euch noch nicht installiert sind, könnt ihr sie unter Ubuntu wie folgt installieren.
  - python3 installieren: `sudo apt install python3`
  - pip (Python Paketmanager) installieren: `sudo apt install python3-pip`
  - pytest installieren: `python3 -m pip install pytest`

  Ihr könnt diese Tests mit dem Befehl `make test` ausführen. Dieser Befehl erzeugt eine `.so`-Datei, die dann von der Testumgebung geladen wird. Der Befehl führt alle vorhandenen Test aus. Da dies unter Umständen unübersichtlich sein kann, könnt ihr auch einzelne Funktionen testen. Dazu könnt ihr pytest manuell aufrufen und nach bestimmten Tests filtern. Wenn zum Beispiel nur die Tests für `mkdir` getestet werden sollen geht das mit `make test_mkdir`. Die Namen aller Tests findet ihr im Verzeichnis `tests`. Falls ihr eine andere python-Version bzw. einen anderen Befehl zum starten benutzt als `python3`, dann ändert bitte im `Makefile` in Zeile 23 und 26.

  Für die Funktion `fs_export` steht kein Test zur Verfügung. Ihr könnt euch selbst Tests dazu schreiben oder die Funktion testen, indem ihr das Programm tatsächlich

[1] http://valgrind.org/

ausführt und die Funktion nutzt. Dazu steht euch die Datei `SysProgFiles.fs` zur Verfügung. Ladet diese Datei mit eurem Programm und sucht darin nach einer Bilddatei. Wenn ihr diese auf euer Hostsystem exportiert könnt ihr diese betrachten.

Abgabe:

- **Verpackt** die Datei *src/operations.c* zu einem zip-Archiv mit dem Namen *submission.zip*. Header-files (**.h*) sollen nicht verändert und damit auch nicht mit abgegeben werden. Hierfür könnt ihr das make target *make pack* verwenden. Damit *make pack* funktioniert muss das tool *zip* auf eurem System installiert sein! Das Archiv könnt ihr dann auf ISIS hochladen.
- **Wichtig:** Es ist nicht notwendig das Archiv und dessen Inhalt mit eurem Namen bzw. Matrikelnummer zu personalisieren. Die Abgabe wird automatisch eurem ISIS Account zugeordnet!