Betriebssysteme Labor: Dokumentation der Aufgaben 1 und 2

Teammitglieder: Marvin Falk, Ljubomir Iliev, Nico Fuchs

Aufgabe 1: InMemory File System

Aufgabe 2: On-Disk File System

Betriebssysteme Dokumentation

Inhaltsverzeichnis

1. InMemory File System

1. Aufgabenstellung

Im ersten Teil des Labors sollten wir ein In-Memory File System mit FUSE implementieren. FUSE ermöglicht uns, den Dateisystem-Treiber aus dem Kernel-Mode in den User-Mode zu verlagern. Zuerst soll mit dem Starten des Mountpoints ein leeres Dateisystem erstellt werden. Alle Dateien, die in dem Dateisystem abgelegt werden, sollen im RAM-Speicher des Rechners gehalten werden und nicht auf der Festplatte. Nach dem Beenden des Programms oder Neustart des Rechners etc. müssen die Dateien nicht mehr vorhanden sein, da sie ja nur im Hauptspeicher gespeichert werden sollen.

2. Lösungsansatz und Umsetzung

Zu Beginn haben wir uns mit den verschiedenen Fuse-Funktionen befasst, um ein Bild über die benötigten Variablen und Funktionen zu bekommen. Danach haben wir die Datenstrukturen für unser Dateisystem definiert: Als Metadaten verwenden wir Dateiname, Dateigrösse, Benutzer/Gruppen-ID, Zugriffsrechte, atime, mtime, ctime und ein char Array für die Daten der Datei. Das erste Problem dabei war, wie ein freier Dateiplatz definiert ist. Wir haben uns entschieden, den leeren String “\0” auch als freien Platz für eine Datei zu definieren. Eine freier Platz kann somit mit aufrufen von findFileByName(“\0”) gefunden werden. Danach haben wir die wichtigsten Funktionen implementiert, um eine Datei erstellen, beschreiben und lesen zu können. Das waren fuseGetattr(), Mknod(), readdir() , write() und read(). Nachdem das einfache Schreiben einer Datei funktioniert hat, wurden noch die übrigen Methoden einzeln implementiert und getestet. Um noch übrige Fehler oder undefiniertes Verhalten zu finden nutzen wir die bereits vorhandenen Compliance-Tests und selbst geschriebene Unit-Tests.

3. Funktionen

1. fuseGetattr()

Mit der getattr()-Funktion werden die Metadaten einer Datei oder des Ordners in den Status Buffer geladen. Wenn der Pfad "/" lautet, es sich also um den Ordner handelt wird nur n_links=2 gesetzt, um auch wieder aus dem Ordner kommen zu können (".."). Ansonsten wird der Datei-Index mit findFileByName() gesucht und die Metadaten der Datei in das gegebene statbuffer geladen. Der Rückgabewert ist entweder 0 bei erfolgreichem Laden der Metadaten oder eine -ERRNO-Fehlermeldung, wie z.B. -2, wenn die Datei nicht existiert.

2. fuseMknod()

In dieser Funktion wird eine Datei mit einem bestimmten Namen und deren Zugriffsrechte erstellt. Zuerst wird geprüft, ob die Datei schon existiert. Falls nicht, wird anschließend ein freier Platz im Dateien Array mit findFileByName gesucht. Dann wird die Datei in die MyFsFileInfo mit dem Namen und allen Zugriffsrechten gespeichert. Der Rückgabewert ist 0 bei Erfolg oder -ERRNO bei Fehlern.

3. fuseUnlink()

Mit dieser Funktion wird eine Datei gelöscht. Zuerst wird die Datei gesucht und dann geprüft, ob die Datei leer ist. Falls nicht, werden die Daten der Datei im Speicher gelöscht und der Name auf "\0" gesetzt. Rückgabe Wert ist bei Erfolg 0 oder -ERRNO bei Fehlern.

4. fuseRename()

Mit dieser Funktion wird der Name einer Datei verändert. Falls der neue Name bereits von einer anderen Datei belegt ist, wird diese mit fuseUnlink gelöscht. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

5. fuseChmod()

Mit dieser Funktion werden die Zugriffsberechtigungen einer Datei verändert. Zuerst wird nach der Datei gesucht und danach werden die Zugriffsberechtigungen wie gegeben, verändert und die Zeit des letzten Zugriffs aktualisiert. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

6. fuseChown()

In dieser Funktion wird die Benutzer-/ Gruppen-ID verändert. Es wird der Datei Index anhand des Namens gesucht und die UID und GID der Datei gesetzt. Anschließend wird die Zeit der letzten Modifikation aktualisiert. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

7. fuseTruncate()

In dieser Funktion wird die Größe einer Datei verändert. Wenn die neue Größe kleiner als die alte ist, werden Bytes entfernt. Wenn die neue Größe größer als die alte ist, sind die neuen Bytes zufällig. Es wird zuerst die neue Größe der Datei eingestellt und dann werden so viele Bytes, wie angegeben, neu allokiert (realloc). Anschließend wird die Zeit der letzten Modifikation in den Metadaten aktualisiert. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

8. fuseOpen()

In dieser Funktion wird eine Datei für Schreiben und Lesen geöffnet. Es wird die letzte Zugriffszeit geändert und auch die Counter für die geöffneten Dateien um 1 erhöht. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

9. fuseRead()

In dieser Funktion wird eine Menge von Bytes von einer gegebenen Position und Offset gelesen. Mithilfe eines memcpy werden im gegebenen Buffer die zu lesenden Bytes gespeichert. Das Offset bestimmt die Anfangsposition. Der Rückgabewert ist die Zahl der gelesenen Bytes, ansonsten -ERRNO.

10. fuseWrite()

In dieser Funktion wird eine Menge von Bytes an einer gegebenen Position geschrieben. Zuerst wird die Größe der Datei um die Größe der zu schreibenden Bytes erhöht. Dazu wird mehr Speicher allokiert für die Daten der Datei. Mithilfe eines memcpy werden die neuen Daten aus dem Buffer in die Daten der Datei an der gegebenen Position kopiert. Zum Schluss wird noch die Zeit der letzten Änderung und der letzten Statusaktualisierung aktualisiert. Der Rückgabewert ist die Zahl der geschriebenen Bytes, ansonsten -ERRNO.

11. fuseRelease()

In dieser Funktion wird eine Datei geschlossen. Es wird nach der Datei gesucht und danach, falls diese existiert, wird der Counter für geöffnete Dateien um 1 verringert. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

12. fuseInit()

In dieser Funktion wird das Dateisystem initialisiert. Es wird eine Logfile geöffnet. Diese Funktion wird immer aufgerufen, wenn das System gemountet wird. Falls das Logfile nicht existiert, wird eine Meldung ausgegeben. Sonst wird auf der Konsole ausgegeben, dass das System gestartet ist. Der Rückgabewert ist immer 0.

13. fuseReaddir()

In dieser Funktion wird ein Verzeichnis gelesen. Mithilfe einer For-Schleife wird über alle Dateien iteriert. Freie Datei Plätze werden mit einer if-Abfrage übersprungen, da der Name != “\0” sein muss. Im Buffer werden die Namen der Dateien im Verzeichnis gespeichert. Der Rückgabewert ist bei Erfolg 0, ansonsten -ERRNO.

14. fuseDestroy()

In dieser Funktion wird das Dateisystem bereinigt/zerstört. Nach erfolgreichem Beenden dieser Funktion ist das Dateisystem ausgehängt.

4. Helferfunktionen

1. findFileByName()

In dieser Funktion wird eine Datei nach ihrem Namen gesucht. Mit einer for-Schleife wird über alle Dateien iteriert. Falls die Datei existiert, wird der Index zurückgegeben, falls nicht eine -1.

2. On-Disk File System

1. Aufgabenstellung

Im zweiten Teil des Labors sollen wir ein On-Disk-File-System implementieren. Im Gegensatz zum In Memory FS sollen auch nach Programmende und Neustart des Rechners alle Dateien vorhanden sein. Dies wird durch die Verwendung einer Containerdatei realisiert. Die Funktionen, die implementiert werden sollen, sind dieselben, aber mit verschiedenen Funktionalität.

2. Lösungsansatz

Zuerst haben wir die Dateistrukturen festgelegt (s. 3.). Nach der Festlegung dieser wurden verschiedene Helferfunktionen für die Interaktion mit diesen und der Containerdatei festgelegt, wie bspw. readStructures.

Wie bei der ersten Aufgabe wollten wir erst die wichtigsten Funktionen implementieren, also writeDataBlock(), fuseRead(), fuseWrite() und reallocBlocks(). Nachdem das Dateisystem ohne Neustarten somit wieder funktioniert hat, haben wir uns um das einlesen der Containerdatei gekümmert und schließlich wieder die verschiedenen Metadaten Funktionen und Synchronisierungsmethoden zur Hilfe implementiert

3. Datenstrukturen

1. MyOnDiskFsFileInfo

Dieser Struct enthält die Metadaten der Datei. Hierzu gehören wie schon beim InMemoryFS die Größe der Datei. Anstatt einem Buffer wird hier aber der Index des ersten Datenblocks auf dem BlockDevice festgehalten. Wenn die Größe der Datei 0 ist, besitzt sie auch keine Datenblöcke und der erste Block (firstBlock) wird auf -1 gesetzt. Dies ist hilfreich, da in den Helferfunktionen immer geschaut wird, ob der Blockindex sich im positiven Zahlenbereich (>=0) befindet und so Fehler leichter erkennbar macht.

2. OpenFile

Wird zum Zwischenspeichern des zuletzt gelesenen Blocks verwendet. Sie enthält einen BLOCK_SIZE großen Char-Buffer und den Index des Blocks. Der Buffer wird nur angelegt, wenn vorher die Datei mit fuseOpen() geöffnet wurde. Mit fuseRelease() kann die Datei wieder freigegeben/geschlossen werden.

3. SuperBlock

Der SuperBlock ist der erste Block auf Speichermedium und enthält Informationen zum Dateisystem. Er speichert den Namen des Dateisystems und die Menge an Datenblöcken. Am wichtigsten ist hierbei die Speicherung der Positionen und Größen der verschiedenen Abteile auf dem Speichermedium. Beispielsweise ist dmapBlock der Index des ersten Blocks der DMAP und dmapSize ist die Menge an Blöcken, die für die DMAP reserviert sind

4. Dmap

Enthält nur ein Array namens blockSet. Die Länge des Arrays bemisst sich an der Anzahl der vorhandenen Datenblöcke.

5. FAT

Enthält ein Array namens nextBlock. Dieses Array hat die selbe Länge wie Dmap, da diese auch an der Anzahl der Datenblöcke bestimmt wird. Jeder Eintrag zeigt immer auf den nächsten Block der Datei. Wenn der Folgeblock -1 ist, hat die Datei keine weiteren Blöcke mehr, das heißt die Datei ist hier zu Ende.

4. Funktionen

Die fuseFunktionen haben die gleichen Rückgabewerte wie bei dem InMemory File System (s. oben) und wurden nur hier nicht nochmal aufgeführt. Kurz gesagt, für Fehlerbehandlung wird 0, oder die Anzahl gelesener/geschriebener Bytes (fuseRead/fuseWrite) bei Erfolg oder die negative Fehlernummer (-ERRNO) zurückgegeben.

1. fuseGetattr()

Holen der Metadaten einer Datei oder des Rootordners:
Bei einer Datei werden hier die verschieden statbuf Einträge einfach mit den Werten im jeweiligen MyOnDiskFsFileInfo struct gefüllt. Für das Finden der Datei wird die Methode findFileByName() verwendet, welche dann den Index des benötigten structs zurückgibt .
Zum inMemoryFS hat sich hier nur die Ergänzung weiterer Metadaten aus dem SuperBlock für den Ordner geändert.

2. fuseMknod()

Erstellen einer neuen Datei:
Sucht mithilfe der Methode findFileByName() nach einem freien Dateiplatz und speichert die geänderte Dateiinfo auch im Container mit der Methode updateFileMetaDataOnDisk() ab.

3. FuseUnlink()

Löschen einer Datei:
Ändert den Namen der Datei zu \0 und iteriert über die FAT-Map beginnend mit dem ersten Block, um so alle Blöcke innerhalb der DMAP freizugeben (also setzt diese auf false) und setzt FAT Einträge auf -1. Zum Schluss werden die Methoden updateDmapOnDisk(), updateFatOnDisk() und updateFileMetaDataOnDisk() aufgerufen, um die Änderungen auf der Festplatte festzuhalten.

4. fuseRename()

Umbenennen einer Datei mit einem bestimmten Namen in einen neuen Namen:
Wie InMemory wird hier nur der Dateiname geändert und fuseUnlink aufgerufen falls der Name belegt ist. Neu ist, dass nach den Änderungen mit updateFileMetaDataOnDisk() die Containerdatei synchronisiert wird.

5. fuseChmod()

Ändern Datei Berechtigungen:
Wie InMemory nur aktualisieren der Berechtigungen, aber nach den Änderungen wird mit updateFileMetaDataOnDisk() auch die Containerdatei synchronisiert.

6. fuseChown()

Ändern des Datei Besitzers:
Wie InMemory nur ändern der Gruppe und Nutzer (GID/UID), aber nach den Änderungen wird mit updateFileMetaDataOnDisk() auch die Containerdatei synchronisiert.

7. fuseTruncate()

Verkleinern einer Datei:

Ruft einfach die Helfermtethode “reallocBlocks” auf, welche die benötigten Blöcke berechnet und auch DMAP und FAT entsprechend aktualisiert.

8. fuseOpen()

Öffnen einer Datei:
Es wird nach dem Datei-Index mit findFileByName() gesucht und in einem filehandle dieser gespeichert. Dieser wird in fuseRead() und fuseWrite()

verwendet, damit nicht immer nach der Datei gesucht werden muss. Hier wird auch der Speicher für das openFile-Buffers für das Einlesen der Blöcke allokiert.

9. fuseRead()

Lesen von einer Datei:
Hier wird die Anzahl zu lesender Blöcke und die Anzahl zu überspringender Blöcke (offset) berechnet. Zum Lesen der Blöcke wird die Methode readOpenFileBlock() verwendet, die den openFile-Buffer füllt, der dann in den Rückgabebuffer kopiert werden kann. Beim Lesen des letzten Blocks muss aufgepasst werden, dass nicht zu viel gelesen wird und beim Lesen der ersten Blocks muss das Offset beachtet werden. Kopiert wie schon bei InMemory alles in einen Buffer und gibt die Anzahl der gelesenen Bytes zurück.

10. fuseWrite()

Schreiben einer Datei:
Wenn die Datei größer wird, dann wird die reallocBlocks Helferfunktion aufgerufen, um dort möglicherweise mehr Blöcke zu reservieren und die Datei Metadaten zu aktualisieren. Darauf wird die Anzahl der zu überspringenden Blöcke berechnet. Wenn dies alles bekannt ist, kann mit dem Beschreiben der Blöcke begonnen werden. Beim Ersten kann es ein inBlockOffset geben und/oder die zu schreibende Größe kleiner als Größe eines Blockes sein. Beim Letzten muss nur geschaut werden, dass nicht zu viel überschrieben wird. Zum Überschreiben eines Blockes muss der Buffer im OpenFile-Struct mit der readOpenFileBlock()-Funktion gefüllt werden.

11. fuseRelease()

Schließen einer Datei:
In dieser Funktion wird der OpenFile-Buffer im Speicher freigegeben.

12. fuseInit()

In dieser Funktion wird das Dateisystem initialisiert. Wird beim Starten ein Container auf dem Speichermedium gefunden, wird readStructures() aufgerufen, sonst wird die neue Containerdatei mit CalculateAndSetBlockPositions() initialisiert.

13. fuseReaddir()

Lesen des Ordners:
Diese Funktion gibt in einem gegebenen Buffer die Namen der Dateien im Ordner zurück, wie bei InMemory.

14. fuseDestroy()

Aufräumen des Dateisystems:
Löscht die angelegten Structs vom Dateisystem wieder und unmounted das Dateisystem.

5. Helferfunktionen

1. findFileByName()

Wie schon beim InMemoryFS sucht diese Funktion einfach nach dem Dateinamen. Sie kann auch verwendet werden, um eine noch nicht belegte Datei zu finden. In diesem Fall muss dann nach \0 gesucht werden.

2. readStructures()

Wenn beim Neustarten des Dateisystems eine Containerdatei gefunden werden kann, wird diese Methode aufgerufen. Es wird erst der SuperBlock eingelesen. Danach sind die Positionen der DMAP und FAT Tabelle bekannt und somit können diese auch eingelesen werden. Hierfür wird erst ein Char-Buffer angelegt und dann mit der Methode readBlocks gefüllt. Nachdem der Speicher für die jeweiligen Strukturen allokiert wurde, wird dieses buffer dann mithilfe von memcpy in die Strukturen kopiert.

3. calculateAndSetBlockPositions()

Wenn beim Neustarten des Dateisystems keine Containerdatei gefunden werden kann, wird diese Methode aufgerufen. Hier wird die gewünschte Menge an Datenblöcken angegeben und daraus die Größe der DMAP, FAT und des gesamten Containers berechnet. Der SuperBlock wird mit den berechneten Daten gefüllt.

4. writeBlocks()

Diese Funktion durchläuft eine einfache For-Schleife zum Schreiben einer Menge an aufeinanderfolgenden Blöcken.

5. writeDataBlock()

Diese Funktion schreibt den Buffer in den gegebenen dataBlockIndex mit dem Block Device. Die Hauptfunktion ist das Addieren des Offsets im Datenblock zuständig. Wie viel addiert werden muss, steht im SuperBlock unter dmapBlock.

6. readOpenFileBlock()

Diese Funktion füllt den Zwischenspeicher Buffer der geöffneten/gegebenen Datei mit dem gegeben Blockindex.

7. readFileMetaDataOnDisk()

Diese Funktion liest das struct MyOnDiskFsFileInfo von der ContainerDatei in den Hauptspeicher vom gegebenen Dateindex in files Array.

8. updateFileMetaDataOnDisk()

Diese Funktion schreibt die Metadaten der Datei (MyOnDiskFsFileInfo) in den jeweiligen Block in der Containerdatei des Hauptspeichers.

9. getNextFreeBlock()

Diese Funktion sucht ab dem gegebenen Block nach einem neuen freien Block in der DMAP (also blockSet[block]==false) und gibt diesen zurück.

10. updateDmapOnDisk()

Diese Funktion synchronisiert die Container-Dmap mit der Hauptspeicher-Dmap. Es wird also die gesamte DMAP iterativ in die ContainerDatei geschrieben.

11. updateFatOnDisk()

Diese Funktion synchronisiert die Container-FAT mit der Hauptspeicher-FAT. Es wird also die gesamte FAT-Tabelle in die ContainerDatei geschrieben.

12. reallocBlocks

Ändern der Dateigröße
Berechnet die benötigten Blöcke der Datei:
Falls mehr Blöcke benötigt werden: Mithilfe von getNextFreeBlock() einen freien Block finden und FAT Tabelle entsprechend aktualisieren. Das Suchen der Blöcke ist effizient, da innerhalb des Vorgangs jeder Block nur einmal nachgeschaut wird, ob dieser frei ist und für den nächsten Block wird gleich beim Folgeblock weiter gesucht.
Falls weniger Blöcke benötigt werden, muss über die bleibende Blöcke iteriert werden und restliche Blöcke wie bei Unlink freigeben werden. Hierbei muss auch die Änderung in FAT und Dmap vorgenommen werden.

6. Programmausführung und Testfälle

Beim Testen war die Schwierigkeit erst das Medium Sized Files (1500 Bytes) zum Laufen zu bekommen, da es mehrere Blöcke waren. Ein weiteres Problem war es, wenn die Datei offen geblieben ist, dass auch der openFile Buffer beim Überschreiben aktualisiert werden musste. Bei einer großen Datei musste die Truncate()-Funktion mehrfach aufgerufen werden, sodass nicht zu viele oder zu wenige Blöcke allokiert werden.