

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Ziel . . . . .	2
1.2	Technische Umgebung . . . . .	2
1.3	Struktur der Containerdatei . . . . .	2
1.4	Projektstruktur . . . . .	2
<b>2</b>	<b>Implementierung</b>	<b>3</b>
2.1	Superblock . . . . .	3
2.2	DMAP . . . . .	3
2.3	FAT . . . . .	3
2.4	ROOT . . . . .	4
2.5	FilesystemIO . . . . .	5
2.6	Persistenz . . . . .	6
2.7	Pufferung . . . . .	6
<b>3</b>	<b>Optimierungen</b>	<b>6</b>
3.1	FAT Performance . . . . .	6
<b>4</b>	<b>Hall of Bugs</b>	<b>7</b>
4.1	-1 = 65536? . . . . .	7

# 1 Einleitung

## 1.1 Ziel

Als Ergänzung zu den Inhalten der Vorlesung "Betriebsysteme" und zur Vertiefung des Systemnahen Programmierens unter C++, wurde das Dateisystem MyFs erstellt.

## 1.2 Technische Umgebung

Da Dateisystemverwaltung auf Kernel Ebene stattfindet, wurde File System In User Space (Fuse) als Schnittstelle zwischen dem Kernel und MyFs benutzt. Somit stand auch das Zielbetriebsystem fest, es wurde Linux aufgrund der besseren Handhabung von Fuse unter Linux. Entwickelt wurde das Dateisystem mit Hilfe von XCode unter MacOS in C++. Zur Versionsverwaltung und zur Projektverwaltung wurde GitHub benutzt.

## 1.3 Struktur der Containerdatei

Die Containerdatei wurde intern in 5 Abschnitte unterteilt.

1. Superblock  $\Rightarrow$  *Informationen über das Dateisystem*
2. DMAP  $\Rightarrow$  *Übersicht freier Datenblöcke*
3. FAT  $\Rightarrow$  *Verlinkung von Datenblöcken zu Dateien*
4. Root  $\Rightarrow$  *Informationen über Dateien*
5. Datenblöcke  $\Rightarrow$  *Dateien in MyFs geteilt in Blöcke*

## 1.4 Projektstruktur

Bei der Projektplanung haben wir uns an dem MVC Pattern orientiert. Wir haben getrennte Klassen für die einzelnen Abschnitte (siehe 1.3), welche die Logik/Modell stellen. Dazu gibt es die Klasse "FilesystemIO" welche Methoden für das Schreiben in die Containerdatei, sowie das Lesen daraus zur Verfügung stellt und in dem Pattern als View angesehen werden kann. Objekte dieser Klassen werden von einer übergeordneten Klasse "MyFS" erzeugt. Diese ruft Methoden aus den Klassen auf, um die Ausführung des Programms zu steuern und kann als Control in dem MVC Pattern angesehen werden.

## 2 Implementierung

### 2.1 Superblock

Der Superblock enthält wichtige Informationen über das Dateisystem an sich. In ihm werden sowohl Informationen die für das Betriebssystem und den Benutzer relevant sind wie Größe des Dateisystems, der freie Speicherplatz und maximale Speichergröße. Als auch für den Betrieb des Dateisystems wichtige Daten wie die jeweiligen Startadressen der Bestandteile und die Größe dieser. Von der Implementierung handelt es sich um eine Struct mit 16 bit unsigned Integer werten.

### 2.2 DMAP

Die DMAP ist dazu da, einen Überblick zu schaffen, welche Datenblöcke frei sind. Im Endeffekt handelt es sich um einen Integer Array dessen Länge, der Anzahl vorhandenen Datenblöcke im Dateisystem entspricht. Dabei bedeutet eine "0" an Stelle x im Array, dass der x-te Datenblock frei ist, dementsprechend eine "1", dass der Datenblock belegt ist.

Die Klasse DMAP stellt 5 öffentliche Methoden, es können Blöcke als belegt und frei gesetzt werden. Außerdem gibt die Methode "getFreeBlock" den nächsten freien Block zurück. Wir haben uns dazu entschlossen, den Wert für den nächsten freien Block jederzeit im Hintergrund als private Variable zu halten und erst nachdem dieser Block beschrieben wurde, den neuen nächsten freien Block zu berechnen. Für die Initialisierung, wurden zwei Methoden implementiert "getAll" welchen den kompletten DMAP Array zurückgibt, und "setAll" welche den DMAP Array mit dem übergebenen Array überschreibt. Diese zwei Methoden werden benutzt, um die komplette DMAP in die Containerdatei auf der Festplatte zu schreiben bzw. sie wiederherzustellen.

### 2.3 FAT

Die Aufgabe der FAT ist die Verbindung zwischen einzelner Datenblöcke zu einer Datei herzustellen. Die FAT realisieren wir als einen Array. Die Größe entspricht dabei der Menge an Datenblöcke die unser Dateisystem zur Verfügung hat. In unserem Fall 65535. Somit ist gewährleistet, dass zu jeder Datei ein Index in der FAT vorhanden ist. Die Verknüpfung erfolgt dabei indem der Wert des Arrays an einer bestimmten Stelle den Index des nächsten Datenblocks beinhaltet. Hinzu kommt noch ein Terminierendes Zeichen, welches zeigt, dass die Datei zu ende ist, bei uns erfüllt diese Aufgabe der Index 65535 da dieser außerhalb des Arrays liegt. Zum auslesen der FAT wurde

eine Methode "getNext" implementiert, diese bekommt den Index des letzten bekannten Datenblocks übergeben und liefert den Eintrag an dieser Stelle im FAT Array, also den Index des nächsten Datenblocks. Zum Löschen einer Datei aus der FAT wird nur der erste Datenblock an die Methode "deleteFromFAT" übergeben. In dieser wird durch die Verkettung iteriert und alle zusammenhängende Einträge gelöscht. Zum Einfügen neuer Informationen in die FAT stellt diese zwei verschiedene Methoden zur Verfügung. Zum einen "addNextToFAT" dieser werden zwei Indizes übergeben, die Methode stellt eine Verkettung zwischen den ersten und dem zweiten Datenblock her. Und die zweite Methode "addLastToFAT", welche an den übergebenen Index den Terminalen Index schreibt.

## 2.4 ROOT

Im ROOT Abschnitt werden Dateien spezifische Informationen gespeichert wie zum Beispiel der Name, letzter Änderungszeitpunkt, Index des ersten Blocks, Größe der Datei und weitere Metadaten. Gespeichert werden die Informationen in einem Array, dessen Größe abhängig von der Maximalen Anzahl Dateien in dem Dateisystem ist, sodass es zu jeder möglichen Datei genau einen Index im Rootarray gibt. Als Datentyp des Arrays haben wir von uns geschriebenes Struct "filestats" gewählt. Zum Auslesen der Informationen stellt die Klasse eine Methode "get" welche überladen ist, einmal kann die Methode mit dem Parameter Dateiname und einer leeren fileStats Struct aufgerufen werden und einmal mit dem Parameter Index im ROOT Array und einer leeren fileStats Struct dabei werden jeweils die Korrekte Struct aus dem Array gefunden, entweder durch Vergleich des Dateinamens oder direkt durch die Position im Array und in die übergebene leere Struct reingeschrieben. Des Weiteren wurden Methoden für zum Löschen von Einträgen, zum Überschreiben einer FileStat im Array, eine überladene Methode zum Erstellen neuer leerer Einträge dabei muss der Dateiname eingegeben werden, die Eingabe der Lese/Schreibrechte ist optional. Es besteht auch die Möglichkeit zu Überprüfen, ob ein Array Eintrag unter dem übergebenen Index vorhanden ist, als auch zum Auslesen des Dateinamens. Eine wichtige Methode stellt dabei die "set" Methode dar, ihr wird ein Index im Array und der Pfad zu einer Datei übergeben. Die Methode liest die Eigenschaften der Datei unter dem gegebenen Pfad aus und speichert diese in einer FileStat unter dem gegebenen Index ab. Hinzu kommen Methoden zum Überprüfen ob ein Eintrag unter dem übergebenen Index existiert und eine extra Methode zum Auslesen des Dateinamens aus einem bestimmten Index.

## 2.5 FilesystemIO

Da in unserem Dateisystem selten Daten einzeln vorliegen, sondern meistens in komplexeren Datenstrukturen wurde eine extra Klasse geschrieben, mithilfe welcher eine komplette Datenstruktur auf die Festplatte geschrieben werden kann. Die Klasse FilesystemIO stellt zwei generische Methoden, eine zum lesen von der Festplatte und eine zum schreiben. Die Parameter sind jeweils die Blocknummer ab welcher gelesen bzw. geschrieben werden soll, die Datenstruktur an sich und die Größe der Datenstruktur. Die Methoden funktionieren ähnlich, deswegen wird aus Gründen der Lesbarkeit die Funktion anhand der Schreibmethode erklärt.

Übergabeparameter sind:

1. block  $\Rightarrow$  *erster Block der Containerdatei welcher beschrieben werden soll*
2. data  $\Rightarrow$  *Pointer auf die zu schreibende Datenstruktur*
3. size  $\Rightarrow$  *Größe der Datenstruktur*

Da die Methode generisch mit verschiedenen Datenstrukturen funktioniert, wird die Datenstruktur zuallererst in einen char pointer gecastet. Anschließend wird ein char Buffer der genau die Größe eines Blockes besitzt. Mithilfe der übergebenen Größe der Datenstruktur wird berechnet wie viele Blöcke zu beschreiben sind. Dann kommt der eigentliche Schreibvorgang welcher in zwei Abschnitte unterteilt wurde, dem ersten, bei dem in einer For-Schleife die Bytes aus der Datenstruktur in den Buffer kopiert werden und dieser mittels der gegebenen Blockdevice Klasse auf die Festplatte geschrieben. Dies geschieht, bis nur noch der letzte Block übrig bleibt da nicht gewährleistet ist, dass genau so viele Bytes übrig sind wie in einen Block passen. Deshalb wird der letzte Block separat geschrieben, indem zuerst mithilfe der Größe der Datenstruktur modulo der Blockgröße berechnet, wie viele Bytes noch übrig bleiben, diese werden dann in den Buffer geladen und auf die Festplatte geschrieben.

## 2.6 Persistenz

Um sicherzustellen, dass das Dateisystem Persistent ist, wird in regelmäßigen Abständen das komplette Dateisystem auf der Festplatte gesichert, für diesen Zweck wurden in den Klassen FAT, ROOT und DMAP, die Klassen "getAll" und "setAll" implementiert. Die

## 2.7 Pufferung

Um viele Lesevorgänge zu vermeiden wurde ein Puffer implementiert, dieser beinhaltet jeweils den zuletzt gelesenen Block einer geöffneten Datei. Als Puffer fungiert eine Struct welche die Blocknummer als auch die Daten selbst speichert. Dabei besitzt jede geöffnete Datei Ihren eigenen Puffer. Diese werden in der MyFs Klasse in einem Array verwaltet. Bei jedem Lesevorgang wird überprüft, ob der Block im Puffer ist, wenn nicht wird der Block eingelesen und der Puffer aktualisiert.

# 3 Optimierungen

## 3.1 FAT Performance

Die ursprüngliche Idee für das Auslesen war es, die komplette Verkettung als Liste wiederzugeben daher wurde eine Methode implementiert, welche den Index des ersten Datenblocks und eine leere Liste übergeben bekommt. Die Methode geht durch die komplette Verkettung durch und speichert die Indizes in der mit übergebenen Liste chronologisch ab. Dies hat sich jedoch beim Testen als sehr langsam und für unsere Zwecke nicht optimal herausgestellt. Als Lösung des Problems wurde ein anderer Ansatz gewählt. Die neue Methode bekommt den letzten bekannten Index übergeben und liefert den nächsten verketteten Index. Dieser Ansatz hat sich als deutlich Performanter herausgestellt, vor allem für größere Dateien. Nachdem selben Prinzip wurde auch die Methode zum hinzufügen neuer Datenblöcke der Verkettung überarbeitet. Auch dies brachte einen großen Geschwindigkeitsvorteil.

## 4 Hall of Bugs

### 4.1 -1 = 65536?

Ursprünglich war es gedacht den kompletten *uint16<sub>t</sub>* Adressraum auszunutzen. Sprich dass die FAT / DMAP  $2^{16}$  Adressen zur mögliche Einträge zur Verfügung haben. Als terminierendes Zeichen war die -1 angedacht. Dies führte dazu, dass aus -1 in der unsigned Darstellung die 65536 wurde und da wir den kompletten Adressraum ausnutzen wollten, war 65536 eine valide Adresse. Zeigen tat sich dies indem sehr oft auf den letzten Block geschrieben wurde obwohl eigentlich die Datei zu ende sein sollte. Die erste Idee war, wir vergrößern den Adressraum und benutzen *uint32<sub>t</sub>*. Dies hatte jedoch zur Folge, dass die größe der FAT auf 512 Blöcke wuchs und somit größer war als alle anderen Abschnitte zusammen. Also war das auch nicht die optimale Lösung. Als elegant und mit den wenigsten Einbußen empfanden wir die Lösung, den *uint16<sub>t</sub>* Adressraum zu benutzen bloß von vorne rein einen Block weniger zu verwenden und diesen dann als terminierendes Zeichen zu benutzen. Somit sinkt unser Effektiv genutzter Speicherraum um einen Block, dies ist jedoch hinzunehmen im Vergleich zu der Verschwendung der Blöcke im Falle des *uint32<sub>t</sub>* Adressraum.