Betriebssysteme
Übung 3, Ergebnisbericht

Tillmann Faust

18.3.2025

Inhaltsverzeichnis

1 Einleitung

1.1 Plattform

Alle Programme wurden für Ubuntu geschrieben und in einer Virtuellen Maschine getestet. Für die Python Implementierungen wurde Python 3.10 verwendet, die Java Implementierungen wurden in Java 21 geschrieben.

1.2 Vorüberlegungen

Erste Überlegungen mit ZFS und den Interaktionen mit dem Dateisystem haben gezeigt, dass das Erstellen von ZFS-Filesystem in einem existierenden Ordner dafür sorgt, dass der ursprüngliche Inhalt für die Dauer der Existenz des Filesystem überschattet werden. Gleiches gilt, wenn man ein Filesystem in einem Ordner erstellt, der Teil eines anderen Filesystems ist. Es wurde sich daher dazu entschieden das erstellen und Löschen von Ordnern ebenfalls über den Dateimanager laufen zu lassen, um nicht jedes mal wenn ein Filesystem erstellt oder gelöscht wird Dateien verschieben zu müssen und darüber den Überblick behalten zu müssen.
Bei der Frage, wie das Parallele Bearbeiten von Dateien umgesetzt werden soll gibt es zwei primäre Möglichkeiten:

In-Place. Bei einer In-Place Bearbeitung wird einem Nutzer beim Öffnen einer Datei der gesamte Inhalt übermittelt, welchen er dann speichern oder im Ram halten, manipulieren und anschließend zurücksenden muss. Diese Methode hat den Vorteil, dass keine zusätzlichen Dateien vom Manager angelegt und verwaltet werden müssen, allerdings muss stattdessen der Inhalt der Datei vollständig in den RAM geladen werden. Ein weiterer Nachteil ist, dass der Nutzer zum zwischenspeichern entweder selbst eine Datei anlegen muss, oder mehrere kleine Commits auf die Originaldatei ausführen muss.

Copy and Overwrite. Die alternative Möglichkeit ist es, bei jeder Dateiöffnung eine Kopie der Datei in einem gesammelten Ordner zu erstellen auf welche der Nutzer freien Zugriff hat. Sobald der Nutzer mit der Bearbeitung fertig ist kann er einen Commit einleiten, welcher bei Erfolg den Inhalt der geöffneten Datei mit dem Inhalt der Kopie überschreibt und die Kopie löscht. Diese Methode erlaubt es einem Nutzer nicht den Gesamten Inhalt permanent zu laden und er kann Änderungen an der Datei speichern, ohne einen Commit ausführen zu müssen. Für diese Methode wurde sich letztendlich entschieden.

2 Dateimanagementsystem: transaction_daemon

2.1 Allgemeines

Als Programmiersprache für den ersten Prototypen wurde Python 3.10 gewählt. Python ist zwar vergleichsweise langsam, was es für Systemnahe Anwendungen ungeeignet macht, jedoch bietet es eine gewisse Übersichtlichkeit und Einfachheit, was es zumindest für einen ersten proof of concept qualifiziert. Geplant war anschließend zuerst die einzelnen Module in C zu übersetzen und in die Python-Software zu integrieren, bevor das gesamte Projekt dann in C

übersetzt worden wäre. Zuletzt hätte noch das gesamte Projekt, oder zumindest der Transaction-, Conflict- und Filesystem-Manager zu einer Datei zusammengefasst werden können um unnötige Dopplungen von Codesegmenten und Methoden-/Klassenaufrufen zu vermeiden.
Ziel war es das System nach Fertigstellung als Daemon im Hintergrund auf dem Rechner, bzw. der VM, laufen zu lassen. Es soll beim Hochfahren des Rechners, bzw. der VM gestartet werden, im Hintergrund auf Anfragen warten und diese Bearbeiten. Es ist anzumerken, dass die Softare nicht umfänglich getestet ist. Zwar wurde sichergestellt, dass die Ausführung der Methoden, sowie Konflikterkennung zumindest in einfachen Situationen funktioniert, ein In-Depth-Testing fand jedoch nicht statt. Insgesamt gibt es leider eine Menge an Problempotential im System.
Die Abgabe des transaction_deamons beinhaltet ein Python-Environment 'transaction_daemon/env', und die Hauptdatei run.py beginnt mit einem Shebang, welches auf dieses Environment verweist. Es sollte theoretisch nicht nötig sein selbst irgendwelche zusätzlichen Installationen vorzunehmen. Es ist notwendig das Programm mit sudo Rechten zu starten, da der FilesystemManager diese für Interaktionen mit dem Dateisystem und ZFS benötigt.

2.1.1 Architektur

Das System besteht aus zwei Hauptbestandteilen: einer API, welche die Interaktion zwischen Nutzer und System ermöglichen soll und der Transaktionslogik des Systems, welche wiederum aus den drei Modulen filesystem_manager, conflict_manager, und transaction_manager besteht. Weiterhin gibt es das Modul exceptions für alle Anwendungsspezifischen Fehler, welche auftreten können und ein Modul utils, in welchem sich eine Logger-Klasse befindet, welche jedoch nicht sinnvoll implementiert ist und bisher nur zum print-debugging verwendet wurde.

2.1.2 Funktionsweise

Aufgrund der Beobachtungen in den ersten Experimenten mit ZFS wurde sich dazu entschieden den Dateimanager als eine Interaktionsebene zwischen Nutzer und ZFS zu designen, welche bevorzugt vom Nutzer verwendet werden soll. Das heißt, dass alle wichtigen Operationen auf dem Dateisystem (das Erstellen, Bearbeiten und Löschen von Dateien, sowie das Erstellen und Löschen von Ordnern) laufen idealerweise über den Dateimanager. Nutzer mit sudo-Rechten können zwar trotzdem Änderungen vornehmen, diese Änderungen könnten jedoch in Konflikt mit dem Dateimanager kommen.

2.2 API

Die Interaktion zwischen einem Nutzer und dem Dateimanagementsystem findet über eine interne API statt. Der erste Prototyp des Projekts nutzt das System einen internen Flask-Server, welcher unter der Adresse "http://localhost:5000" um die Kommunikation zwischen Nutzern, z.B. JZfsWrapper-Instanzen, und dem System zu ermöglichen. Geplant war es, nach einer erfolgreichen Implementation von der Flask Web-API zu einem Unix Domain Socket zu wechseln, da diese schneller sind und eine web-API für die Anwendung außerhalb vom Prototyping ungeeignet ist. Bei erfolgreicher Anfrage gibt die API einen Code

200 mit einer Erfolgsmeldung und den für den Nutzer wichtigen Daten zurück, Bei Misserfolg wird entweder ein Code 400 mit einem zusätzlichen Fehlercode zwischen 400 und 408 und einer kurzen Fehlerbeschreibung zurückgegeben oder ein Code 500, wenn es sich um einen nicht speziell abgefangenen oder unerwarteten Server-internen Fehler handelt. Die Codes 400-408 entsprechen keinem Standard und sind, bis darauf, dass sie 400ter Codes sind von den etablierten HTTP-Codes (z.B. Error 404) zu unterscheiden.

Rückmeldecodes

Code	Bedeutung
200	Erfolg
400	Die Anfrage ist fehlerhaft
401	Die UUID existiert nicht
402	Die Transaktion ist entweder invalide oder hat nie existiert
403	Der Dateipfad ist keine Datei oder kann nicht gefunden werden
404	Der Dateipfad existiert bereits
405	Der Pfad kann nicht gefunden werden
406	Der Pfad existiert bereits
407	Der Pfad kann keinem Filesystem zu- oder untergeordnet werden
408	Der Pfad kann wegen offener Transaktion nicht entfernt werden
500	Interner Server Fehler/Catch-All Fehler

2.3 transaction_manager

Der TransactionManager im Modul transaction_manager stellt das zentrale Verwaltungssystem dar. Er verwaltet die UUIDs aktiver Nutzer, erstellt Transaktionen und verwaltet diese mithilfe des ConflictManager und setzt Änderungen am Dateisystem mithilfe des FilesystemManager um. Jede Methode der TransactionManager mapped auf eine routing-Methode der API im Modul app/routes.

2.4 filesystem_manager

Das Modul filesystem_manger und die zugehörige Klasse FilesystemManager bilden die unterste Ebene der Transaktionslogik und umfassen die Logik zur Interaktion zwischen Dateimanager und dem ZFS-Dateisystem. Da sowohl der ConflictManager, als auch der TransactionManager auf verschiedene Methoden des FilesystemManager zugreifen wurde dieser als Singleton implementiert. Bei erster Initialisierung werden alle aktuell existierenden Filesystems abgefragt und in den Listen 'self._top_level_fs' und 'self._filesystems' abgespeichert. Die erste Liste dient als Referenz um zu prüfen, ob ein angegebener Pfad als Filesystem instantiiert werden kann, die zweite speichert alle Filesystems zur späteren Referenz und Verwendung. Durch diese Umsetzung hat der Manager jedoch nicht zugriff auf alle ZFS Pools auf dem System, sondern nur die Teile von Pools, welche über ein Filesystem verfügen. Dies ist eine Konsequenz aus dem Missverständnis, dass angenommen wurde, dass das erstellen eines FS auf dem einem Ordner in einem existierenden FS die Ordner-Unterstruktur beibehält und nicht

mit einer neuen Struktur überschattet. Der Code wurde dahingehend noch nicht überarbeitet. Die Methoden der Klasse sind:

Methode	**Beschreibung**
get_fs	Nimmt einen Pfad gibt den zugehörigen ZFS-Filesystem Namen zurück, sofern dieses existiert
make_fs	Erstellt ein Filesystem, welches den angegebenen Pfad im System darstellt. Schlägt fehl, wenn der neue Pfad kein Unterpfad eines existierenden Filesystems ist
destroy_fs	Entfernt ein Filesystem und alle in ihm liegenden Dateien. Vernichtet alle Dateien, wenn diese nicht anderweitig gesichert sind
create_snapshot	Erstellt einen ZFS-Snapshot
restore_snapshot	Stellt einen Snapshot wieder her und entfernt alle Snapshots Welche nach dem wiederhergestellten vom selben Filesystem erstellt wurden.
destroy_snapshot	Löscht einen ZFS-Snapshot
create_file_copy	Erstellt eine Dateikopie im Ordner 'self._filecopy_path'
delete_file_copy	Löscht die angegebene Dateikopie
create_file	Erstellt eine Datei, sofern der angegebene Pfad einem Filesystem entspricht
delete_file	Entfernt die angegebene Datei, sofern diese existiert und in einem Filesystem liegt
overwrite_file	Überschreibt die Datei im origin_path mit der Datei im copy_path

2.5 conflict_manager

Der ConflictManager ist für das erkennen und beheben von Konflikten zuständig. Erreicht wird dies, indem für jedes aktuell verwendete Filesystem eine sogenannte Konfliktliste geführt wird, welche offene Transaktionen mit der zugehörigen Datei, einem Snapshot des Filesystems und einem Timestamp assoziiert und so bei jedem Commit prüft, ob dieser für einen Konflikt sorgt. Zum Speichern und Verwalten der Einzelnen Transaktionen wird die Klasse ConflictNode verwendet. Da der ConflictManager Snapshots erstellen, löschen und das System auf diese zurücksetzen können muss, verfügt er ebenfalls über eine Referenz auf die FilesystemManager-Instanz.

2.5.1 Konflikterkennung

Die Konflikterkennung erfolgt über eine Reihe von Listen, welche jeweils mit einem Filesystem, also einem Pfad, assoziiert werden. Wann immer eine Transaktion gestartet wird, so wird eine entsprechende ConflictNode erstellt und gespeichert. Das starten einer Transaktion selbst führt zu keinem Konflikt, kann aber für Konfliktpotential sorgen. Wenn ein Nutzer eine Dateiänderung umsetzen möchte so wird eine entsprechende ConflictNode erstellt und an zeitlich korrekter Stelle eingefügt. Anschließend wird geprüft, ob zwischen dem Start und Commit der Transaktion bereits ein anderer Commit auf derselben Datei statt fand. Wenn ja, so liegt ein Konflikt vor, das Filesystem muss auf den

Punkt vor Transaktionsstart zurückgesetzt werden und alle CommitNodes zwischen und inklusive Start- und Commit-Knoten entfernt werden. Wenn nein wird zusätzlich geprüft, ob zwischen Start und Commit eine Transaktion auf derselben Datei gestartet wurde, wenn ja, so gibt es immer noch Konfliktpotential und der Commit Knoten darf noch nicht entfernt werden. Wenn eine Transaktion abgebrochen wird, werden alle Commit-Knoten, welche zwischen dem zugehörigen (inkl.) und dem nächsten (exkl.) Start-Knoten liegen entfernt. Diese Commits gelten als konfliktfrei umgesetzt.

3 Java Bibliothek: JZfsWrapper

Die Java Library JZfsWrapper dient, wie der Name andeutet, als Wrapper-Bibliothek für API Anfragen an den Dateimanager. Sie besteht aus einer Reihe von anwendungsspezifischen Exceptions, der Klasse JsonParser und der Hauptklasse TransactionHandler.

3.1 TransactionHandler

Der TransactionHandler stellt die Schnittstelle zwischen Nutzer und Dateimanager dar. Über ihn können Nutzer Dateien erstellen, löschen, öffnen, comitten, sowie offene Transaktionen abbrechen. Weiterhin können neue Pfade, also eigentlich ZFS-Dateisysteme, erstellt oder existierende entfernt werden. Die Klasse registriert sich bei Instantiierung über die Methode getTransactionManager() automatisch beim Dateimanager und speichert die UUID intern. Wenn die API nicht verfügbar ist, oder es einen Serverfehler gibt, so wird stattdessen 'null' zurückgegeben. Um dem Nutzer möglichst viel Verwaltungsaufwand zu sparen findet die Interaktion zwischen Handler bzw. API und dem Nutzer nur über String und java.io.File Objekte statt. Bevor ein Programm, welches einen TransactionHandler verwendet beendet wird, sollte die close()-Methode aufgerufen werden, welche wiederum die Methode deregister aufruft. Diese Methode soll den Nutzer wieder von der API deregistrieren. Geschieht dies nicht so findet ebenfalls ein deregister()-Aufruf in der Methode finalize() statt, welche durch den Garbage-Collector aufgerufen wird, allerdings ist dies kein sicherer Weg, da es passieren kann, dass die jvm geschlossen wird, bevor der GC die Möglichkeit hat die Methode auszuführen. In diesem Fall wird der Dateimanager mit unbenutzten UUIDs etwas 'zugemüllt'.

Methoden

Methode	Parameter	Return	Beschreibung
openFile	String path	File file	Öffnet eine Datei im angegebenen Pfad und gibt ein File-Objekt mit der Dateikopie zurück
commitFile	File file	Boolean success	Versucht die Änderungen in der angegebenen Datei im Dateisystem umzusetzen
cancelFile	File file	Boolean success	Bricht die aktuelle Dateibearbeitung ab ohne einen Commit zu machen
createFile	String path	Boolean success	Versucht den angegebenen Pfad als Datei zu erstellen
removeFile	String path	Boolean success	Versucht die angegebene Datei zu entfernen
createDirectory	String path	Boolean success	Versucht den angegebenen Pfad als zfs-Filesystem zu erstellen
removeDirectory	String path	Boolean success	Versucht den angegebenen Pfad zu entfernen

Da der Dateimanager mit Transaktions-IDs arbetet werden die erstellten File-Objekte intern mit der zugehörigen TID durch eine HashMap assoziiert. Die Fehlermeldungen, welche bei der Bearbeitung der API-Anfragen durch den Dateimanager entstehen können werden in den entsprechenden Methoden abgefragt und anschließend durch Exceptions an den Nutzer weiter gegeben. Die geworfenen Exceptions sind:

Exceptions

Methode	Exception
openFile	java.io.IOException java.io.FileNotFoundException PathNotManagedException
commitFile	java.io.IOException TransactionInvalidException
cancelFile	—
createFile	java.io.IOException java.io.FileNotFoundException PathNotManagedException PathNotFoundException
deleteFile	java.io.IOException java.io.FileNotFoundException PathNotManagedException
createDirectory	java.io.IOException PathNotManagedException PathExistsException
deleteDirectory	java.io.IOException PathNotManagedException DirectoryInUseException

Nicht-java.io Exceptions entstammen dem Package com.faustens.os24u3.exceptions. Die Exceptions sind Teilweise jedoch nicht eindeutig definiert, beziehungsweise geht eine gewisse Menge Information verloren gehen; so wird beispielsweise bei API-Anfragen sowohl bei einer IOException, als auch bei einer InterruptedException eine IOException geworfen. Neben den genannten wird zusätzlich von jeder Methode eine generische Exception geworfen um weitere Fehler in Java, sowie Internal Server Errors der API abzufangen.

3.2 JZfsException

Die Klasse JZfsException dient als Oberklasse für alle speziell für dieses Projekt geschriebenen Exceptions. Die Exceptions dienen hierbei eher als Messager um Nutzer über den Erfolg von Anfragen und Zustand des Systems zu informieren. Die Funktion der Exceptions sollten durch den Namen bereits größtenteils selbsterklärend sein, seien der Vollständigkeit halber hier dennoch gelistet.

Exception	**Bedeutung**
PathExistsException	Der angegebene Pfad existiert bereits
PathNotFound Exception	Der angegebene Pfad kann nicht gefunden werden
PathNotManagedException	Der Pfad ist nicht Teil eines ZFS-Pools bzw. eines Ober-ZFS-Filesystem
TransactionInvalidException	Die angegebene Datei gehört zu einer Transaktion welche entweder einen Konflikt verursacht hat, durch einen Konflikt invalidiert wurde, oder nie existiert hat
DirectoryInUseException	Der angegebene Pfad, bzw. das Filesystem, kann nicht gelöscht werden, da noch offene Transaktionen in ihm existieren

3.3 JsonParser

Die Klasse JsonParser ist eine Hilfsklasse und stellt drei Methoden zum Generieren und Zerlegen von Json Strings zur Verfügung. parseJsonString versucht den eingegebenen String als JSON-String zu interpretieren und Schreibt die Key-Value-Paare in eine HashMap, mapToJsonString nimmt eine HashMap<String,String> als Eingabe und generiert einen JSON-String und getJsonString nimmt eine variable Anzahl an Strings, prüft ob es sich um eine gerade Anzahl handelt und wandelt die Eingabe in diesem Fall in einen JSON-String um, wobei für $0 \leq i \leq$ args.length/2 immer das $2i$-te Argument einem Key und das $2i + 1$-te Argument dem zugehörigen Value entspricht.

4 Brainstorming Tool: JZfsBrainstormer

Der JZfsBrainstormer war als Brainstorming-Tool konzipiert, welches es Nutzern erlaubt mithilfe des Dateimanagers so genannte Vaults in einem von ZFS verwalteten Filesystem zu erstellen und darin Ideen zu erstellen und zu kommentieren.

4.1 Funktionsweise

Nutzer haben die Möglichkeit mithilfe der Befehle

- 'java -jar JZfsBrainstormer.jar check <path>'
- 'java -jar JZfsBrainstormer.jar make <path>'
- 'java -jar JZfsBrainstormer.jar open <path>'

Ordner darauf zu prüfen, ob sie Vaults sind (erkennbar an einer vorhandenen Datei 'vaultfile'), einen Ordner zu einer Vault zu machen oder eine Vault zu öffnen um dort Ideen zu erstellen oder zu kommentieren. Wird eine Vault geöffnet, so kann der Nutzer folgende Befehle nutzen:

- list: Listet alle vorhandenen Ideen
- open <note_name>: Zeigt den Inhalt und die Kommentare einer Notiz an und erlaubt es einen Kommentar zu dieser Notiz zu verfassen
- create: Erlaubt es eine neue Notiz zu erstellen
- close: Schließt die Vault und das Brainstorming-Tool

4.2 Architektur

Das Werkzeug basiert, wie in der Aufgabe gestellt auf der Bibliothek JZfsWrapper um die Interaktion mit dem Dateisystem durchzuführen. Das Programm besteht aus einer Main Methode, einem VaultManager, welcher für die Interaktion mit dem Vault System zuständig ist, sowie der Klasse Note, welche als Softwareinterne Repräsentation der Notizen dient und Methoden zum Auslesen von und schreiben in Notizdateien bereitstellt. Die Dateien der einzelnen Notizen verwenden XML um Titel, Content und beliebig viele Comments zu speichern.

4.3 Probleme

Beim schreiben der Tools sind leider eine Reihe von vermutlich tiefgreifenden Problemen aufgetreten, deren Finden, geschweige den Lösen leider nicht mehr zeitlich zu machen scheint. Obwohl ein laufender transaction_daemon bei allen versuchten Zugriffen zum erstellen von Pfaden und öffnen von Dateien eine Erfolgsmeldung gibt, werden über die Klasse TransactionManager aus der JZfs-Wrapper Bibliothek Fehlermeldungen zurückgegeben, welche dafür sorgen, dass das Brainstorming-Tool über das Testen, Erstellen und Öffnen von Vaults hinaus keine Interaktion ermöglicht. Dass das Programm überhaupt so weit kommt liegt einzig daran, dass durch den TransactionManager geworfene Fehler ignoriert werden und weitergearbeitet wird, als hätte es diese nicht gegeben, was bei tatsächlichem Auftreten eines Fehlers zu massiven Problemen führt.
Eine Vermutung ist, dass es Probleme mit der Korrekten Interpretation der Rückgabecodes im TransactionHandler gibt, bzw. dass der JsonParser Probleme macht. Dies steht allerdings im Widerspruch zu vorherigen Tests, in welchen jede dieser Teile problemlos funktioniert hat. Ein weiteres Problem könnte eventuell die Ordner oder Package-Struktur der beiden Projekte sein. Ungeachtet dessen bleibt das Tool leider unfertig und in einem Funktionsunfähigen Zustand.

5 Rückblick und Ausblick

Aus zeitlichen Gründen konnte das Projekt leider nicht so weit fertiggestellt werden wie ich es mir gewünscht hätte. Es funktioniert zwar, zumindest auf meiner Maschine, und ist oberflächlich getestet hat jedoch dennoch eine Menge von Macken und Ungereimtheiten die nicht mehr ausgearbeitet werden konnten. Die Bibliothek JZfsWrapper wurde einmal als JAR Kompiliert und testweise in einem erfolgreich Projekt verwendet. Würde das Projekt weiter fortgesetzt werden, dann wären die nächsten Ziele den Dateimanager zu vereinheitlichen; Fehlermeldungen klar zu definieren; die Erkennung von ZFS-Pools und Filesystems zu überarbeiten, so dass das System Zugriff auf alle von ZFS verwalteten Systeme hat; und es muss ein Umstieg von Flask zu Unix System Sockets durchgeführt werden. Weiterhin müssen neue, voll umfängliche Tests ausgeführt werden unter anderem auch mit dem Dateimanager als tatsächlicher System-Daemon und nicht als ein vom Nutzer gestarteten Programm. Zuletzt muss das Programm in eine geeignetere Sprache übersetzt und die Struktur ggf. angepasst werden.

6 Abgabedateien

Die Abgabe erfolgt über Github im Projekt Faustens/OS24U3. Das Projekt umfasst die Ordner transaction_daemon mit dem Dateimanagementsystem, den Ordner JZfsWrapper mit der Java Bibliothek und den Ordner JZfsBrainstormer mit dem unfertigen Brainstorming tool. Weiterhin befindet sich dieser Ergebnisbericht als PDF unter den Dateien. Zur Bearbeitung der letzten Aufgabe war leider keine Zeit mehr.