BERICHT ZUR OPTIMISTISCHE NEBENLÄUFIGKEIT MIT ZFS-SNAPSHOTS

Narek Grigoryan

Matrikelnummer: 1547616

Kurs: Betriebsysteme– Übung 3

Inhalt

Einleitung

Im Rahmen der Übungsaufgabe zu Betriebssystemen wurde die Implementierung eines Dateiverwaltungssystems mit ZFS-Snapshots gefordert. Da macOS keine native Unterstützung für ZFS bietet, musste eine alternative Lösung gefunden werden. Ziel dieses Berichts ist es, die einzelnen Schritte der Einrichtung und Implementierung zu dokumentieren.

Dieser Bericht besteht aus mehreren Teilen:

1. Setup der Entwicklungsumgebung: Hier wird erläutert, wie die ZFS-Unterstützung unter macOS eingerichtet wurde, einschließlich der Installation von OpenZFS und der Konfiguration einer Linux-VM.
2. Entwicklung einer Java-Bibliothek für ZFS-Transaktionen: Diese Bibliothek verwaltet parallele Dateioperationen mit ZFS-Snapshots und stellt sicher, dass Änderungen nur übernommen werden, wenn keine Konflikte auftreten.
3. Prototyp einer Brainstorming-Anwendung: Aufbauend auf der Java-Bibliothek wird eine Anwendung entwickelt, die es mehreren Benutzern erlaubt, Ideen zu speichern und zu bearbeiten, ohne dass dabei Dateikonflikte entstehen.
4. Validierung und Testen: Abschließend wird ein Validierungstool entwickelt, das die Wahrscheinlichkeit von Dateioperationen mit Konflikten maximiert und analysiert.

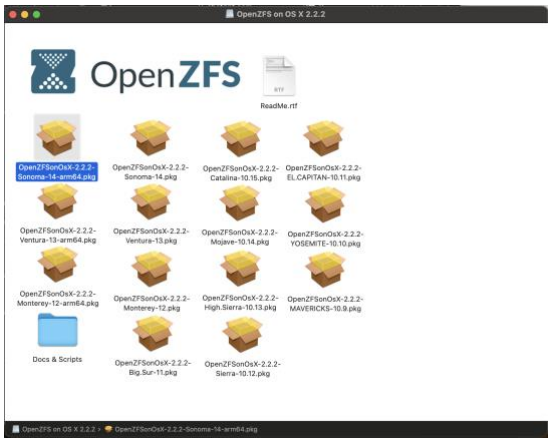
1. Setup der Entwicklungsumgebung

Problemstellung

Da macOS keine native ZFS-Unterstützung bietet, musste eine separate Installation erfolgen. Die Installation über Homebrew funktionierte nicht, sodass die Pakete manuell heruntergeladen und installiert werden mussten.

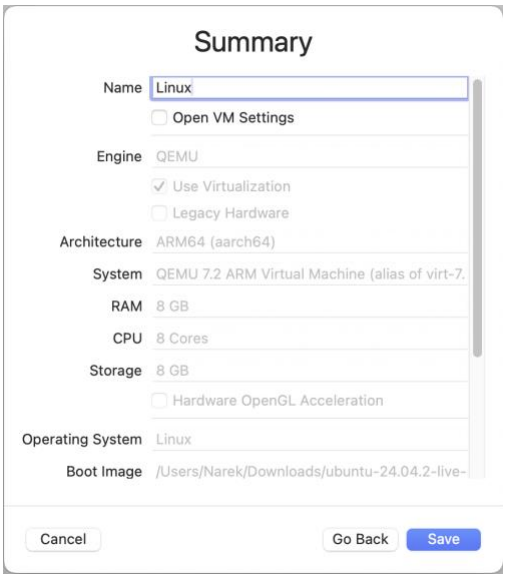
Schritte zur Installation

Zunächst wurde die passende OpenZFS-Version gesucht. Da das MacBook auf einem Apple M3-Chip basiert, musste die arm64-Version von OpenZFS heruntergeladen werden. Die Datei OpenZFSOnOsX-2.2.2-Sonoma-14-arm64.pkg wurde ausgewählt und die Installation gestartet.
Während der Installation trat jedoch ein Problem auf. macOS blockierte die Systemerweiterung mit einer Fehlermeldung: "System Extension Blocked – A program tried to load new system extension(s) signed by Joergen Lundman." Trotz mehrfacher Versuche konnte dieses Problem nicht gelöst werden, weshalb eine alternative Lösung notwendig wurde. Die Entscheidung fiel darauf, eine virtuelle Maschine mit Linux einzurichten, um ZFS in einer nativen Umgebung zu nutzen.

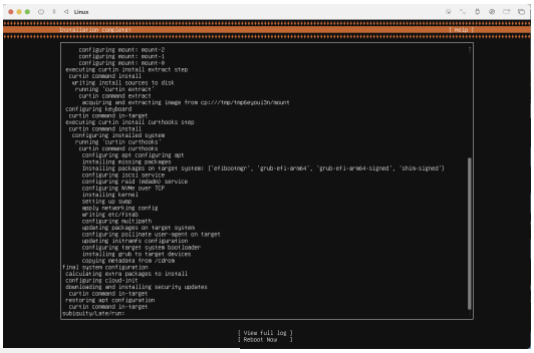


Einrichtung einer Linux-VM mit UTM

Zunächst wurde die neueste Version von Ubuntu (ubuntu-24.04.2-live-server-arm64.iso) heruntergeladen. Anschließend wurde mit UTM eine neue virtuelle Maschine erstellt. Als Engine wurde QEMU verwendet, da es ARM64 unterstützt. Aufgrund der begrenzten Speicherkapazität des MacBooks konnte die VM nur mit 8 GB RAM und 8 CPU-Kernen konfiguriert werden. Der Speicherplatz wurde auf 8 GB begrenzt.



Nach der Konfiguration wurde die Ubuntu-ISO als Boot-Image ausgewählt und die Installation gestartet. Während des Installationsprozesses wurden Partitionierung und Dateisystem-Einrichtung durchgeführt. Danach wurde ein Benutzerkonto erstellt und die Installation abgeschlossen. Nach einem Neustart konnte Ubuntu erfolgreich gestartet und genutzt werden.

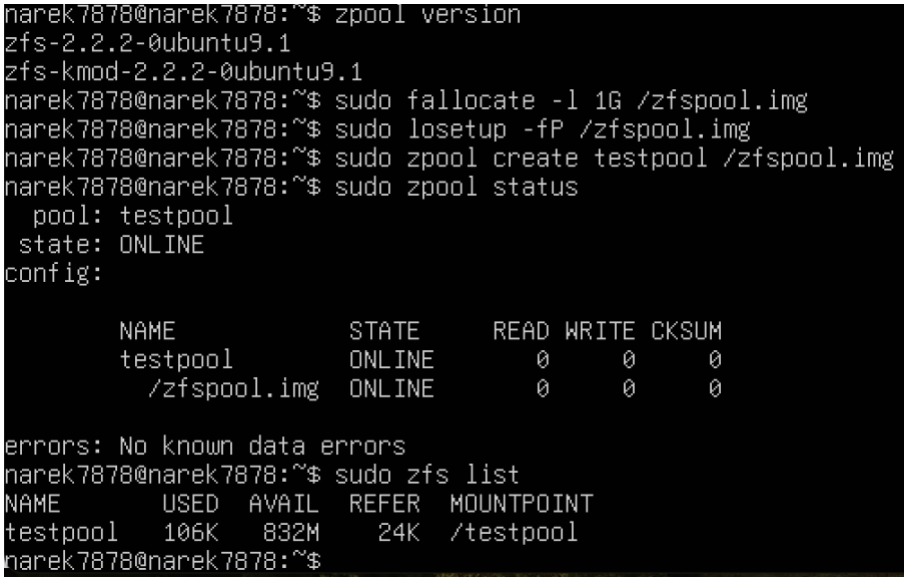


Einrichtung von ZFS in der VM

Nachdem Ubuntu erfolgreich installiert war, wurde ZFS installiert. Zunächst wurde überprüft, ob ZFS korrekt installiert ist. Dazu wurde der Befehl `zpool version` ausgeführt, der bestätigte, dass `zfs-2.2.2-0ubuntu9.1` installiert ist.

Um ZFS zu testen, wurde ein neuer Pool erstellt. Dazu wurde ein 1 GB großes Speicher-Image (/zfspool.img) erzeugt. Anschließend wurde es als virtuelles Laufwerk eingebunden und mit dem Befehl `sudo zpool create testpool /zfspool.img` ein neuer ZFS-Pool mit dem Namen testpool erstellt.

Die erfolgreiche Einrichtung des ZFS-Pools wurde durch den Befehl `sudo zpool status` bestätigt. Der Status des Pools wurde als "ONLINE" angezeigt, was bedeutet, dass er korrekt funktioniert. Zusätzlich wurde mit `sudo zfs list` geprüft, ob das Dateisystem erkannt wurde. Hier wurde testpool als mountbares Dateisystem angezeigt.



2. Java-Bibliothek

2.1 Zielsetzung der Aufgabe

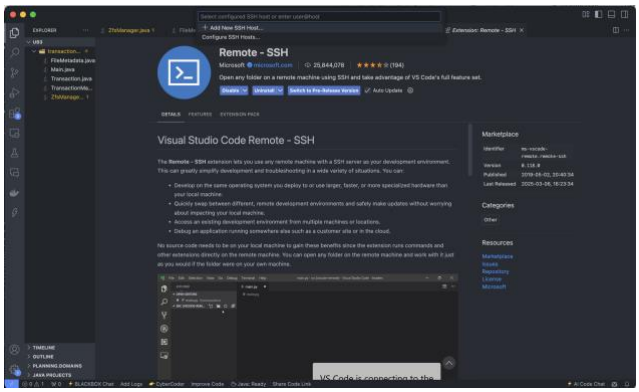
In Aufgabe 1 sollten wir eine Java-Bibliothek konzipieren und implementieren, die folgende Anforderungen erfüllt:

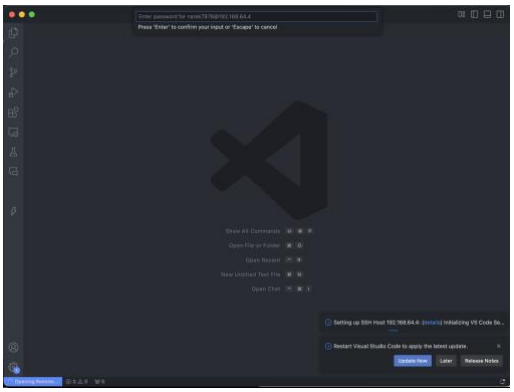
1. **Transaktionslogik bei Dateimodifikationen**:
    - Jede Dateioperation soll in einer Transaktion gekapselt werden.
2. **ZFS-Snapshots programmatisch erstellen und verwalten**:
    - Bei Beginn jeder Transaktion wird ein ZFS-Snapshot erstellt, um später gegebenenfalls ein Rollback durchführen zu können.
3. **Konflikterkennung anhand von Metadaten** (z. B. Hash oder Timestamp):
    - Wenn eine andere Transaktion dieselbe Datei während der Laufzeit verändert hat, wird beim Commit ein Konflikt erkannt → automatischer Rollback.
4. **Grundlegende Dateioperationen** (Lesen, Schreiben, Löschen)
    - Innerhalb einer Transaktion sollen alle Änderungen isoliert und atomar sein.
5. **Isolation & Atomizität**
    - Entweder werden alle Änderungen übernommen (Commit) oder gar keine (Rollback).

2.2 Setup der Umgebung

Im vorherigen Kapitel haben wir die virtuelle Linux-Maschine erfolgreich mit dem UTM-Tool zum Laufen gebracht. Nun musste die Linux-Maschine mit Visual Studio Code verbunden werden, da dort später der Code implementiert wird.

Dafür musste das bereits auf meinem macOS installierte VS Code mit einer **Remote-Assist-Erweiterung** ausgestattet werden, um eine Verbindung zur virtuellen Maschine herzustellen. Zusätzlich musste auf der Linux-Maschine **Java installiert** werden, damit die Java-Bibliothek ausgeführt werden kann.





```
narek7878@narek7878:/$ java -version
openjdk version "21.0.6" 2025-01-21
OpenJDK Runtime Environment (build 21.0.6+7-Ubuntu-124.04.1)
OpenJDK 64-Bit Server VM (build 21.0.6+7-Ubuntu-124.04.1, mixed mode, sharing)
```

Damit steht uns nun eine reine **Linux-Umgebung** zur Verfügung, in der wir **ZFS-Snapshots, ZFS-Rollbacks und alle weiteren ZFS-Befehle** nutzen können. Die Programmierung selbst erfolgt jedoch weiterhin bequem in VS Code auf dem Mac.

2.3 Implementierung der Java-Bibliothek

2.3.1 Methodik und Vorgehen

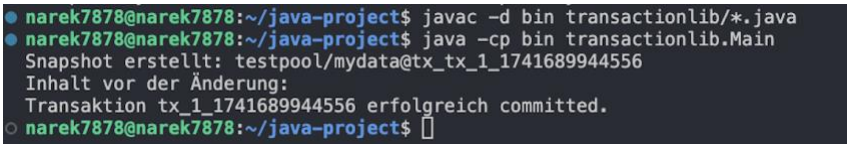
Die Umsetzung orientiert sich an einem **optimistischen** Locking-Ansatz. Das bedeutet:

1. Bei **Transaktionsstart** wird ein ZFS-Snapshot erstellt.
2. Jede Dateioperation (Lesen, Schreiben, Löschen) läuft in einem **temporären Arbeitsbereich** bzw. wird zumindest so vorbereitet, dass wir den ursprünglichen Zustand kennen.
3. Beim **Commit** wird geprüft, ob sich die Dateien seit Beginn der Transaktion geändert haben (Mittels Timestamp/Hash).
4. **Kein Konflikt** → Änderungen werden übernommen (Commit).
5. **Konflikt** → Rollback zum Snapshot (transaktionelle Änderungen verworfen).

2.3.2 Klassen und Zuständigkeiten

- ZfsManager:
  - Erstellt, löscht und rollt ZFS-Snapshots zurück.
  - Arbeitet intern mit ProcessBuilder (z. B. sudo zfs snapshot dataset@tx_X).
- TransactionManager:
  - Verwaltet alle Transaktionen.
  - Erstellt für jede neue Transaktion (z. B. beginTransaction()) eine eindeutige ID und ruft den ZfsManager auf, um den Snapshot zu erstellen.
- Transaction:
  - Kapselt die Dateioperationen der einzelnen Transaktion.
  - Speichert initiale Metadaten (z. B. Timestamp/Hash) jeder Datei in einer Map.
  - Beim commit() prüft sie, ob sich die Live-Dateien verändert haben. Bei Konflikt → rollback().
- FileMetadata:
  - Hilfsklasse zum Ermitteln der Dateimetadaten (Timestamp, SHA-256-Hash).
- Main.java:
  - Ein einfaches Startbeispiel.

- Zeigt, wie man eine Transaktion beginnt, eine Datei liest und schreibt und dann committet.
- Folgende Screenshot zeigt den erfolgreichen Commit



2.3.3 Vorgehen in Code und Bezug zur Aufgabenstellung

1. **Snapshot-Erzeugung:**

   - Beim Aufruf von beginTransaction() erstellt der ZfsManager via zfs snapshot dataset@tx_ID einen neuen Snapshot.
   - Dies gewährleistet Isolation, da wir später im Konfliktfall einfach zum Snapshot zurückkehren.

2. **Konfliktprüfung:**

   - Die Transaktion speichert bei erstem Zugriff (Lesen, Schreiben, Löschen) die ursprünglichen Metadaten.
   - Beim Commit vergleicht sie diese Metadaten mit den aktuellen Live-Daten.
   - Stimmen sie nicht überein (anders als beim Start)? → Konflikt → Rollback.

3. **Dateioperationen:**

   - Lesen (readFile), Schreiben (writeFile), Löschen (deleteFile).
   - Damit wird das Grundgerüst geschaffen, um jede beliebige Anwendung (z. B. Brainstorming-Tool) umzusetzen.

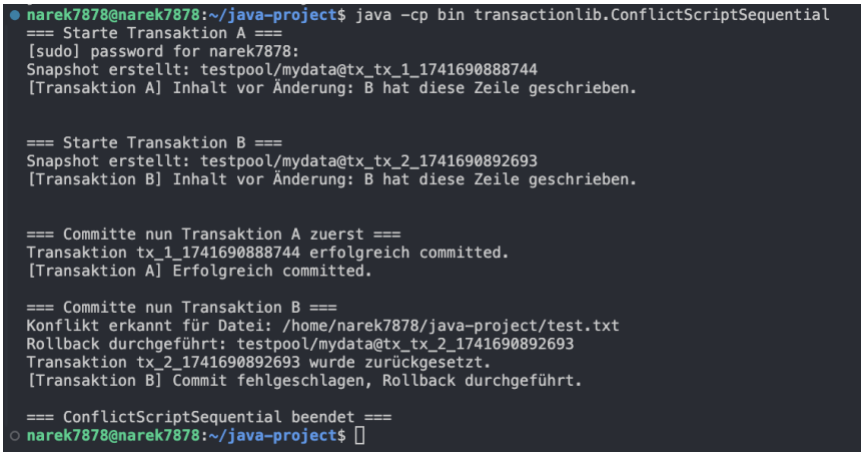
4. **Atomizität & Isolation:**

   - Entweder committen wir alle Änderungen auf einmal oder gar nicht → kein „Zwischenzustand“.
   - Durch Snapshot-basierte Rollbacks bleibt das System konsistent.

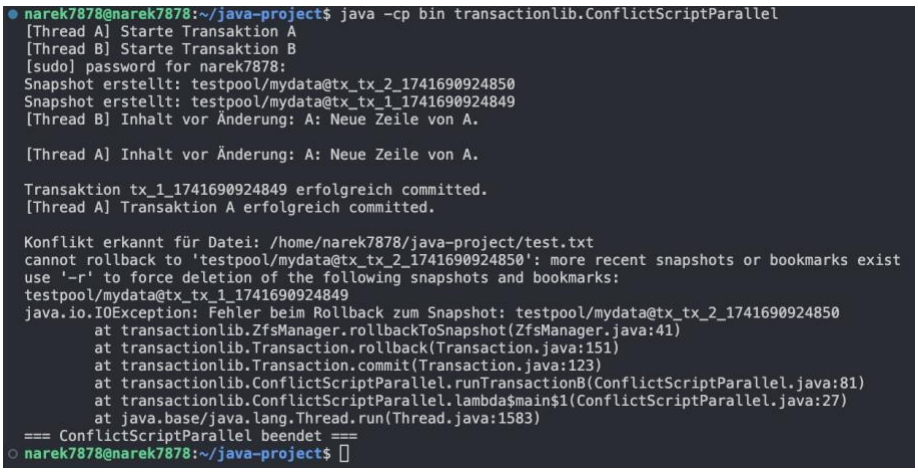
2.3.4 Tests mit Konfliktszenarien

Um sicherzustellen, dass die **Konflikterkennung** und der **Rollback** in realistischen Szenarien funktionieren, wurden **zwei Testskripte** implementiert:

1. ConflictScriptSequential.java:
    - Hier erzeugen wir **zwei Transaktionen (A und B) nacheinander**.
    - B liest einen alten Stand, während A noch nicht committet hat, dann committet A → Live-Datei ändert sich → B commitet später → Konflikt (Rollback).
    - Siehe folgende Screenshot, der zeigt, wie B am Ende in Konflikt gerät und ein Rollback durchführt.



2. ConflictScriptParallel.java:
    - Hier werden zwei Threads (A und B) parallel gestartet.
    - Beide transaktionalen Zugriffe überschneiden sich in der Zeit, was oft zum Konflikt führt.
    - Folgende Screenshot zeigt, dass ein Fehler (cannot rollback to snapshot…) auftreten kann, wenn ZFS einen neueren Snapshot sieht. Trotzdem erkennt das System den Konflikt und möchte ein Rollback machen – die Logik ist korrekt.



Erklärung → ZFS erlaubt standardmäßig **kein** Rollback auf einen älteren Snapshot, wenn zwischenzeitlich schon ein „neueren“ oder „weiteren“ Snapshot gibt, **und** der neuere Snapshot noch existiert. Auch wenn die Namen „tx_1_...“ und „tx_2_...“ suggerieren, dass „2“ zeitlich später ist, kann es für ZFS so aussehen, dass ich zuerst Snapshot B erstellt habe (oder bereits A gecommittet wurde und sein Snapshot weiter existiert). In jedem Fall blockiert ZFS das Rollback, wenn es auf der Timeline eine „fortgeschrittenere“ Version hat, außer man nutzt das Flag -r („recursive“), das evtl. jüngere Snapshots zerstört.

- Also die ZFS-Einschränkung kann man durch `zfs rollback -r` umgehen oder durch seriellere Abläufe.

2.4 Rückblick auf die Implementierung

Aus Sicht der Übungsanforderung ist dieses Vorgehen **meine Meinung nach ausreichend** und demonstriert das geforderte Zusammenspiel aus Snapshot-Erzeugung, Konflikterkennung und dem atomaren Commit bzw. Rollback.

Gleichzeitig ist mir bewusst, dass dieses System **in einer realen Parallel**- und **Mehrbenutzerumgebung** an Grenzen stößt, etwa weil:

1. Ein Rollback stets das gesamte Dateisystem auf den Snapshot zurücksetzt und damit potenziell alle Änderungen nach dem Snapshot (auch von anderen Transaktionen) verwirft.
2. ZFS ein Rollback nur dann zulässt, wenn keine neueren Snapshots in der Timeline existieren (außer man erzwingt es mit `-r`, was dann aber andere Snapshots zerstören würde).
3. Ich die Dateien beim ersten Zugriff aus dem Live-System kopiere, statt sie streng aus dem Snapshot zu holen – was für echte Parallelität eine Lücke sein kann, wenn unmittelbar vor dem Kopiervorgang bereits jemand anders an derselben Datei schreibt.
4. Das Hashen großer Dateien pro Transaktion zu Performanceproblemen führen könnte.

Trotz dieser Punkte erfüllt meine Lösung die Aufgabenstellung, weil sie das Prinzip einer optimistischen Nebenläufigkeit mit Snapshots aufzeigt und in einer kontrollierten (z. B. seriellen) Testumgebung gut funktioniert. ***Für den Echtbetrieb müsste man jedoch aufwendigere Mechanismen einsetzen, zum Beispiel ZFS-Clones, vollständige Pfadverwaltung sowie das konsequente Löschen oder Zusammenführen von Snapshots, um parallele Transaktionen sauber handhaben zu können und Konflikte besser zu isolieren.***

3. Prototyp einer Brainstorming-Anwendung

3.1 Zielsetzung der Aufgabe

Die Aufgabe bestand darin, eine **einfache kollaborative Brainstorming-Anwendung** zu entwickeln, die es ermöglicht, Ideen in einer Datei zu speichern und mit Kommentaren zu versehen. Die Anwendung sollte **optimistische Nebenläufigkeit** mit **ZFS-Snapshots** verwenden, sodass parallele Dateioperationen atomar und konfliktfrei ablaufen. Falls zwei Benutzer gleichzeitig dieselbe Datei bearbeiten, sollte ein Konflikt erkannt und eine der beiden Transaktionen **zurückgesetzt (Rollback)** werden. Die gesamte Implementierung erfolgte als **textbasiertes CLI-Tool**.

3.2 Implementierung der Brainstorming App

3.2.1 Nutzung der bestehenden Transaktionslogik mit ZFS-Snapshots

Da bereits eine Java-Bibliothek für Transaktionsmanagement konnte darauf aufgebaut werden. Die Hauptlogik der Anwendung wurde in
die Klasse BrainstormingApp implementiert, welche folgende Hauptfunktionen enthält:

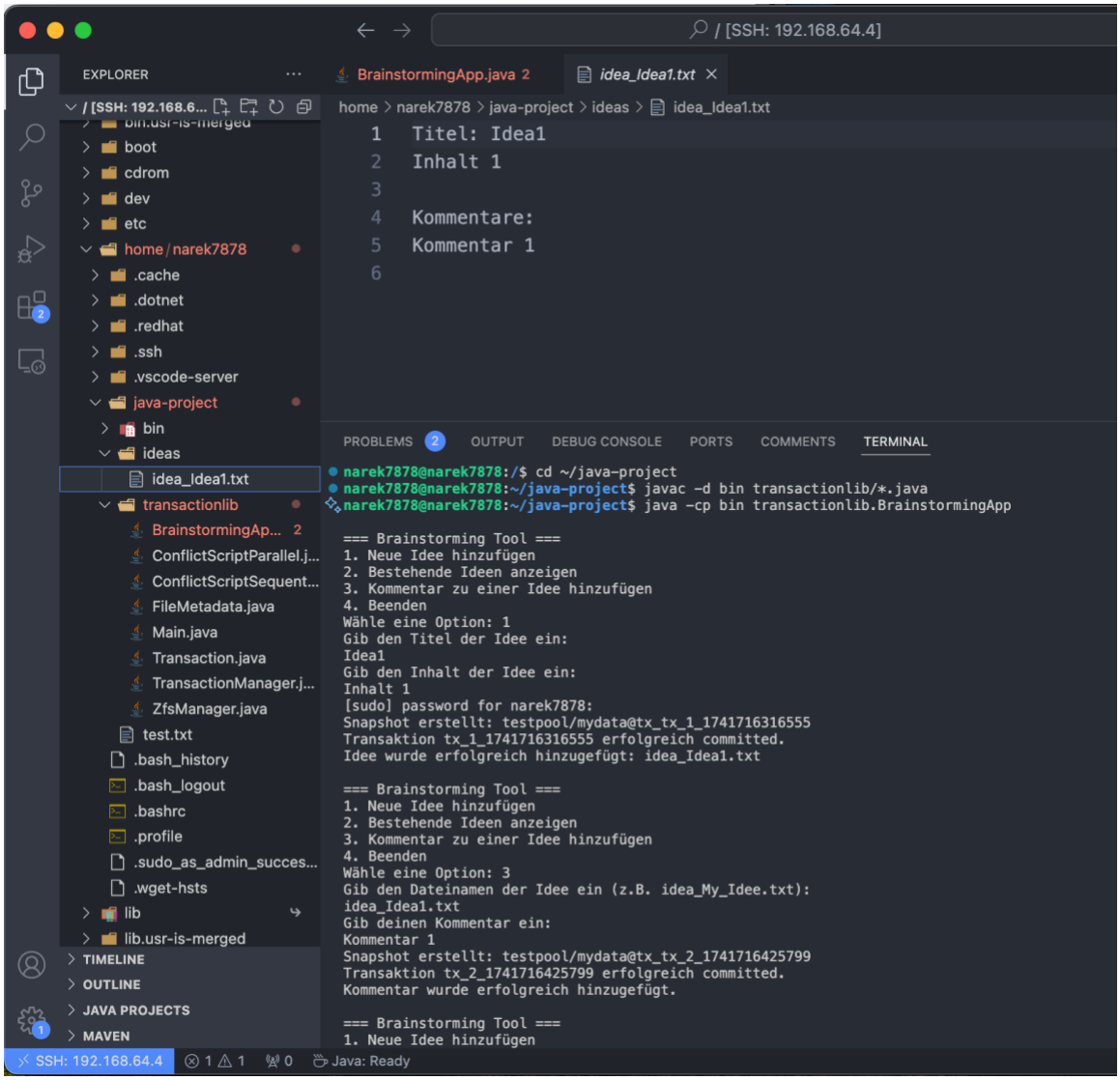
- **Neue Idee hinzufügen**: Erstellt eine Datei für eine neue Idee mit initialem Inhalt und Kommentar-Bereich.
- **Bestehende Ideen anzeigen**: Listet alle vorhandenen Ideen im Verzeichnis auf.
- **Kommentar zu einer Idee hinzufügen**: Öffnet eine Transaktion, liest die bestehende Datei, ergänzt den neuen Kommentar und versucht die Änderungen zu committen. Falls sich die Datei zwischenzeitlich geändert hat, wird ein Rollback durchgeführt.

3.2.2 Funktionsweise der Brainstorming-Anwendung

Um die grundlegende Funktionalität sicherzustellen, wurde die Anwendung manuell getestet. Das erste Screenshot zeigt, dass die Anwendung korrekt arbeitet:

- Neue Idee (idea_Idea1.txt) hinzugefügt.
- Kommentar erfolgreich eingetragen.
- Transaktions-Commit ohne Konflikt.

Im folgenden screenshot wird deutlich, dass die grundlegenden Dateioperationen **reibungslos** ablaufen und das System mit **ZFS-Snapshots**arbeitet, um Änderungen zu sichern.



3.3 Testen der Robustheit mit parallelen Zugriffen

Da eine der zentralen Anforderungen der **optimistische Nebenläufigkeit** ist, wurde ein spezielles Testskript (test_parallel.sh) erstellt, das **mehrere parallele Prozesse** startet und gezielt versucht, die Anwendung unter Stress zu setzen.

3.3.1 Funktionsweise des Skripts test_parallel.sh

Das Skript simuliert zwei parallele Transaktionen:

- Prozess A schreibt einen Kommentar in idea_Idea1.txt.
- Prozess B schreibt gleichzeitig einen anderen Kommentar in dieselbe Datei.
- Beide Prozesse starten nahezu gleichzeitig und versuchen unabhängig voneinander zu committen.

- Falls ein Prozess den Commit zuerst abschließt, erkennt der zweite Prozess einen Konflikt und führt ein Rollbackdurch.

3.3.2 Ergebnisse aus dem Testskript

Das folgende Screenshot zeigt ein Teil der Testergebnisse:

- In jedem Durchlauf konnte eine Transaktion erfolgreich committen, während die andere zurückgesetzt wurde.
- Typische Meldungen im Terminal:
  - Erfolgreicher Commit: „Transaktion ... erfolgreich committed.“
  - Rollback durch Konflikt: „Konflikt erkannt ... Transaktion wurde zurückgesetzt.“
- Das bedeutet, dass die Konflikterkennung und das Rollback-Verhalten korrekt funktionieren.

Der finale Dateiinhalt bestätigt das erwartete Verhalten (orange box im Screenshot).Hier sieht man, dass in jedem Durchlauf nur eine Transaktion erfolgreich war.

- **Warum ist B häufiger erfolgreich als A?**
  - Da das Skript beide Prozesse parallel startet, hängt der Erfolg von mehreren Faktoren ab, insbesondere von der **CPU-Thread-Zuweisung** und der **Prozess-Scheduler-Logik** des Betriebssystems.
  - Moderne Multi-Core-Prozessoren verwenden ein **Thread-Scheduling**, bei dem Betriebssysteme versuchen, Threads effizient auf verfügbare Kerne zu verteilen. Falls B zufällig zuerst auf einem **weniger ausgelasteten CPU-Kern** landet oder eine kürzere Wartezeit auf CPU-Zuweisung hat, kann er schneller den Commit-Vorgang abschließen.
  - Betriebssysteme nutzen zudem **Thread-Priorisierung** und können bestimmte Threads bevorzugen, basierend auf Faktoren wie **Reaktionsfähigkeit und I/O-Zugriffen**. Falls Prozess B vom Betriebssystem bevorzugt behandelt wird (z. B. weil sein Kontextwechsel günstiger ist), hat er eine höhere Wahrscheinlichkeit, vor A zu committen.
  - Der zufällige Effekt hängt daher nicht nur von der Ausführungsgeschwindigkeit der Anwendung ab, sondern auch von **OS-abhängigen Scheduling-Entscheidungen, CPU-Architektur und paralleler Prozesskonkurrenz**.
  - Dies zeigt, dass die Transaktionskontrolle verlässlich funktioniert, aber durch Betriebssystemfaktoren beeinflusst wird, was eine Analyse der Thread-Zuweisung weiter verbessern könnte.

3.4 Rückblick auf die Implementierung

Durch die Implementierung der **Brainstorming-Anwendung mit optimistischer Nebenläufigkeit** wurde ein **robustes System** entwickelt, das **gleichzeitige Dateioperationen ermöglicht**, ohne Daten zu beschädigen. Die **automatisierten Tests mit `test_parallel.sh`** haben gezeigt, dass **Konflikte korrekt erkannt und zurückgesetzt werden**, wodurch die **Datenkonsistenz jederzeit gewährleistet** bleibt.

4. Validierung

4.1 Zielsetzung der Aufgabe

Im Rahmen der dritten Aufgabe wurde das Ziel verfolgt, eine Validierungskomponente in das bestehende Transaktionssystem zu integrieren. Diese sollte gezielt Konflikte zwischen gleichzeitigen Dateioperationen maximieren, um die Robustheit des Systems unter hoher Last zu testen.

Dazu wurde **ValidationTool.java** implementiert, während die übrigen Systemkomponenten bereits in vorherigen Aufgaben entwickelt wurden. Die Validierung sollte automatisiert konkurrierende Schreib- und Leseoperationen erzeugen und die Auswirkungen auf das System analysieren. Wesentliche Metriken wie Konfliktraten, Rücksetzzeiten und Systemdurchsatz sollten dabei berücksichtigt werden.

4.2 Implementierung der Validierungstool

4.2.1 Methodik und Vorgehensweise

Implementierungsansatz

Das Transaktionssystem nutzt **ZFS-Snapshots**, um Änderungen an Dateien zu verwalten. Eine Transaktion erstellt einen Snapshot vor der Modifikation und versucht bei einem Konflikt, darauf zurückzurollen**. ValidationTool.java** erzeugt nun eine große Anzahl paralleler Transaktionen mit zufälligen Schreiboperationen, um gezielt konkurrierende Änderungen herbeizuführen.

Technische Umsetzung

- **Simultane Transaktionsgenerierung**: Das Tool erstellt mehrere parallele Threads, die Schreiboperationen auf eine gemeinsame Datei ausführen.
- **ZFS-Snapshots und Rollbacks**: Jede Transaktion legt einen Snapshot an. Falls eine parallele Transaktion einen Konflikt verursacht, soll ein Rollback durchgeführt werden.
- **Konfliktmaximierung**: Durch Anpassung der Parameter wie Anzahl der Transaktionen oder Schreibhäufigkeitsollten mehr Konflikte erzeugt werden.

4.2.2 Validierungstool (ValidationTool.java)

Die ValidationTool.java-Klasse übernimmt die Aufgabe, eine hohe Anzahl gleichzeitiger Transaktionen zu simulieren und dabei gezielt Konflikte herbeizuführen.

- **Initialisierung**: Das Tool erstellt eine definierte Anzahl von Transaktionen und weist diesen zufällige Schreiboperationen zu.

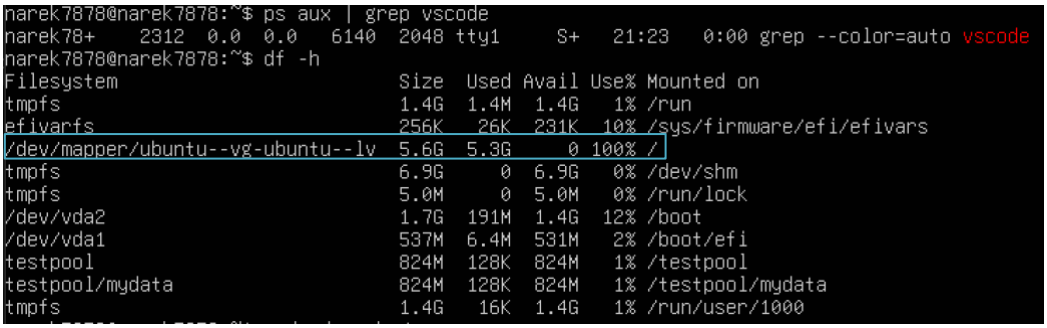
- **Durchführung der Transaktionen**: Diese Transaktionen starten parallel und konkurrieren um die Datei.
- **Erfassung der Ergebnisse**: Am Ende werden Anzahl der erfolgreichen Commits, Rollbacks/Konflikte sowie die durchschnittliche Transaktionsdauer berechnet und ausgegeben.

4.3 Porbleme und Debugging

Speicherplatzproblem durch falsche Parameterwahl

Zunächst gab es kaum Rollbacks, was darauf hindeutete, dass die Konkurrenz zwischen den Transaktionen nicht ausreichend war. Um die Konflikte zu maximieren, wurden die Parameter angepasst. Dabei passierte ein Fehler:

- Statt 100 Transaktionen wurde versehentlich 10.000 Transaktionen ausgeführt.
- Aufgrund der geringen Speicherkapazität der UTM-Linux-VM führte dies zu einem vollen Dateisystem.
- In Screenshot 1 ist zu sehen, dass VS Code nicht mehr kompilieren konnte.
- Screenshot 2 zeigt, dass das Root-Dateisystem zu 100 % belegt war.



Lösung:

- Über das Mac-Terminal konnte noch auf die Linux-VM zugegriffen werden.
- Dateien wurden gesichert, bevor weitere Maßnahmen ergriffen wurden.
- Nach langen verschiednen Ansätzen könnte durch das Entfernen von VS Code-Server-Caches das Problem behoben werden:
- rm -rf ~/.vscode-server
  rm -rf ~/.vscode

- Nach diesem Fix war ein erneuter Zugriff über VS Code Remote möglich.

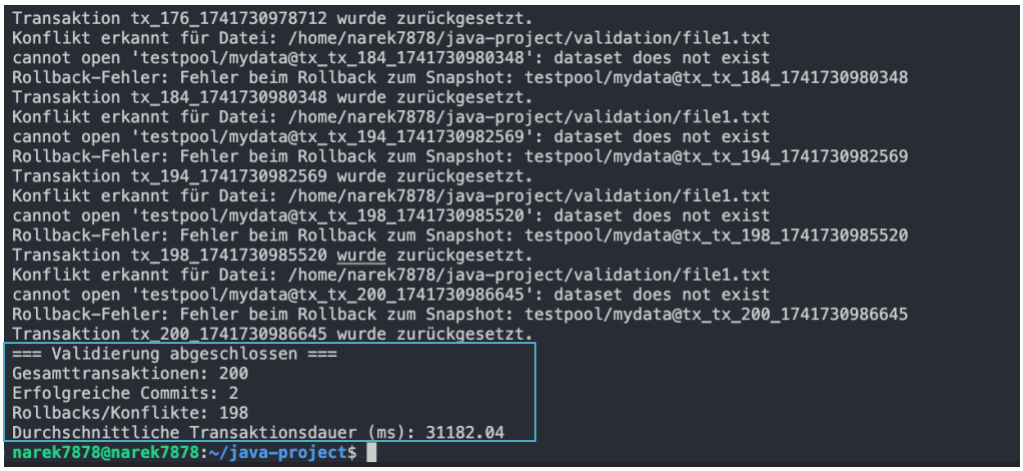
Optimierung der Konflikterzeugung

Nachdem das System wieder funktionierte, wurde die **Konfliktwahrscheinlichkeit weiter erhöht**:

- **Änderungen in ZfsManager.java:**
  - Snapshots wurden mit -r erzwungen, um Abhängigkeitsprobleme zu vermeiden.
  - Verbesserte Fehlerbehandlung für nicht existierende Datasets.
- **Änderungen in Transaction.java**:
  - Reduzierung der Transaktionsdauer, um mehr gleichzeitige Operationen zu erhalten.
  - Mehrfache Dateioperationen pro Transaktion eingeführt.

4.4 Endgültige Validierungsergebnisse

Nach diesen Anpassungen wurde das Validierungstool erneut ausgeführt. In folgenden Screenshot ist zu sehen, dass nun die gewünschten Ergebnisse erzielt wurden:



- Gesamttransaktionen: 200
- Erfolgreiche Commits: 2
- Rollbacks/Konflikte: 198
- Durchschnittliche Transaktionsdauer: 31.182 ms

Damit wurde das Ziel erreicht, ein hohes Maß an Konflikten zu erzeugen und das System unter Last zu testen.

4.5 Rückblick auf die Implementierung

Die Implementierung des Validierungstools hat gezeigt, dass ein transaktionsbasiertes System mit gleichzeitigen Dateioperationen eine Vielzahl von Herausforderungen mit sich bringt. Ein zentraler Aspekt war die Balance zwischen Systemstabilität und der gezielten Erzeugung von Konflikten, um die Belastungsgrenzen und Fehlerbehandlung zu testen.

Während der Umsetzung wurden verschiedene technische Konzepte wie parallele Transaktionsverarbeitung, Snapshot-Management mit ZFS und Konflikterkennung in einem Multi-Threading-Umfeld angewendet. Die Struktur der Implementierung erlaubte es, unterschiedliche Parameter für die Anzahl der Transaktionen, Schreibintensität und Rollback-Strategien anzupassen, wodurch verschiedene Szenarien getestet werden konnten.

Ein wichtiger Erkenntnisgewinn bestand darin, dass die initialen Einstellungen nicht ausreichten, um ausreichend viele Konflikte zu erzeugen. Erst durch gezielte Modifikationen an den Parametern und Anpassungen im Konfliktmanagement konnten realistische Szenarien mit hoher Rollback-Rate erreicht werden.

Trotz der erfolgreichen Umsetzung gibt es Optimierungspotenziale. Beispielsweise könnte die Konfliktwahrscheinlichkeit feiner gesteuert oder zusätzliche Metriken zur Performance-Analyse integriert werden. Insgesamt erfüllt die Implementierung jedoch die Anforderungen der Aufgabenstellung und bietet eine solide Basis für weitere Experimente und Analysen im Bereich der transaktionsbasierten Parallelverarbeitung.