

Ergebnisbericht: Optimistische Nebenläufigkeit mit ZFS-Snapshots

Einführung

In diesem Projekt wurde eine ZFS-Library in Java entwickelt. Zusätzlich wurde ein Brainstorm-Programm zum Testen der Befehle sowie ein Validierungstest erstellt, der die Befehle der Library automatisch ausführt.

Das Programm habe ich auf Ubuntu in einer VirtualBox ausgeführt.

Teil 1: Java-Bibliothek für Transaktionslogik

Funktionsweise

Die ZFS-Library in Java implementiert grundlegende Funktionen wie **Löschen**, **Datei erstellen**, **Datei überschreiben** und **Datei lesen** sowie Hilfsfunktionen wie das Erstellen von Hashwerten.

Der **ZFS-Pool** wird über eine **Konfigurationsdatei** eingerichtet. Diese Datei ermöglicht es dem Nutzer, das Programm oder die Bibliothek individuell an seinen spezifischen Anwendungsfall anzupassen. Damit die Konfiguration erfolgreich angewendet werden kann, muss die Datei dem Programm beim Ausführen beiliegen.

ZFS-Befehle werden über die Methode `executeCommand(String command)` ausgeführt. Diese ermöglicht die Ausführung beliebiger Linux-Terminalbefehle über Java.

```
private static void executeCommand(String command)
    throws IOException, InterruptedException {
    try {
        ProcessBuilder processBuilder =
            new ProcessBuilder("bash", "-c", command);
        processBuilder.inheritIO();
        Process process = processBuilder.start();
        process.waitFor();
    } catch (IOException | InterruptedException e) {
        System.err.println("Fehler beim Ausführen des Befehls: " + command);
        e.printStackTrace();
        throw e;
    }
}
```

ZFS-Befehle und Konfliktbehandlung

Die **ZFS-Befehle** `zfs snapshot` und `zfs rollback` werden zur Dateiverwaltung verwendet, während andere Aspekte über Java implementiert werden.

Konfliktbehandlung

Um eine konsistente Dateiverwaltung mit **optimistischer Nebenläufigkeit** zu gewährleisten, ist es wichtig, mögliche Konflikte zu erkennen und entsprechend zu behandeln. Konflikte treten insbesondere bei Funktionen wie **overwriteFile** auf, die den Inhalt einer Datei verändern.

Konfliktbehandlungsstrategie

Der Ablauf zur Behandlung von Konflikten umfasst folgende Schritte:

1. **Erster Check-In:** Ein Hashwert der Datei wird generiert.
2. **Texteingabe:** Der Nutzer gibt den gewünschten Text ein (in dieser Zeit könnte ein anderer Nutzer die Datei bearbeiten).
3. **Zweiter Check-In:** Vor dem Überschreiben der Datei wird erneut ein Hashwert erstellt.
4. **Check-Out:**
 - **Identische Hashwerte:** Die Datei wurde nicht verändert. Es wird ein Snapshot des aktuellen Zustands erstellt.
 - **Unterschiedliche Hashwerte:** Ein Rollback auf den neuesten Snapshot wird durchgeführt.

Diese Methode stellt sicher, dass die Änderungen des Nutzers erhalten bleiben und die Konsistenz der Datei gewährleistet ist.

Im Code sieht es wie folgt aus (Codesnippet aus Aufgabe 2):

```

    private static void ideeKommentieren(String titel) throws IOException, InterruptedException {
        String dateipfad = IDEEN_VERZEICHNIS + titel + ".txt";
// 1. Check-In
        String originalHash = ZFS_Library.checkIn(dateipfad);
// Nutzer tippt neuen Text
        String newContent = scanner.nextLine();
// 2. Check-In
        String newHash = ZFS_Library.checkIn(dateipfad);
// Datei wird überschrieben
        ZFS_Library.overwriteFile(dateipfad, newContent);
// Check-Out
        ZFS_Library.checkout(originalHash, newHash);
    }

```

Ein Konflikt tritt auf, wenn mehrere Benutzer gleichzeitig Änderungen an derselben Datei vornehmen. Im folgenden Screenshot wird das Szenario dargestellt:

The image shows two screenshots of a terminal window running the Brainstorming-Tool. The window title is 'max@max-VirtualBox: /media/sf_VirtualBox'.

Top Screenshot:

```

max@max-VirtualBox:/media/sf_VirtualBox$ sudo java -jar BrainstormTool.jar

Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
> 4
Titel der Idee: Test
Check-In abgeschlossen. Aktueller Hash: cc34cbf9732f7075e9aed6ec32e4acfd076e72ed4aeb762811afdbc728a2d69
Ich bin der schnellste!
Check-In abgeschlossen. Aktueller Hash: 904f7f42b7877bda9df5541677cdeede5d0e4fdad4f8e16c4b2f841eccc523d7
⚠Konflikt erkannt! Rollback auf den neuesten Snapshot: snapshot_1742330308750
Rollback durchgeführt: snapshot_1742330308750

Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
>

```

Bottom Screenshot:

```

max@max-VirtualBox:/media/sf_VirtualBox$ sudo java -jar BrainstormTool.jar

Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
> 4
Titel der Idee: Test
Check-In abgeschlossen. Aktueller Hash: cc34cbf9732f7075e9aed6ec32e4acfd076e72ed4aeb762811afdbc728a2d69
Ich bin viel schneller!
Check-In abgeschlossen. Aktueller Hash: cc34cbf9732f7075e9aed6ec32e4acfd076e72ed4aeb762811afdbc728a2d69

Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
> 

```

Erklärung des Screenshots

In diesem Beispiel wurde eine Datei zum Bearbeiten geöffnet und parallel in einem zweiten Terminal editiert. Nach Abschluss des Editiervorgangs im ersten Terminal werden die **Hashwerte** überprüft:

- **Oben:** Unterschiedliche Hashwerte – ein Konflikt wird erkannt und ein automatischer Rollback wird eingeleitet.
- **Unten:** Identische Hashwerte – die Datei wurde nicht verändert, und die Änderungen werden erfolgreich übernommen.

Nutzung des neuesten Snapshots

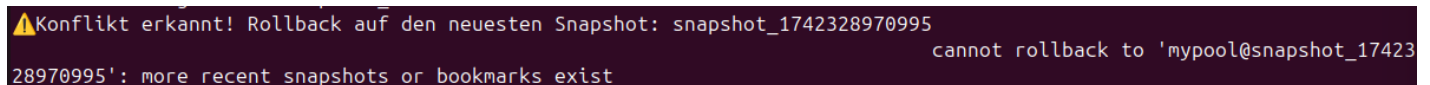
Eine weitere Herausforderung war sicherzustellen, dass stets der **neueste Snapshot** für den Rollback verwendet wird. Um den theoretisch neuesten Snapshot zu ermitteln, werden alle Snapshots anhand von `lastModified` sortiert.

Erkenntnisse aus den Tests

Während der Tests stellte sich heraus, dass in der Zeit, die benötigt wird, um den neuesten Snapshot zu identifizieren, ein neuer Snapshot erstellt werden kann. Dies führt zu folgendem Verhalten:

- ZFS gibt eine Warnung aus, und der Rollback wird verworfen (sofern er nicht explizit erzwungen wird).
- Das System bleibt dadurch immer auf dem aktuellsten Stand.

Der folgende Screenshot veranschaulicht diesen Prozess:



```
⚠ Konflikt erkannt! Rollback auf den neuesten Snapshot: snapshot_1742328970995
cannot rollback to 'mypool@snapshot_1742328970995': more recent snapshots or bookmarks exist
```

Herausforderungen

Eine der anfänglichen Überlegungen war, dem Nutzer die Möglichkeit zu geben, Dateien direkt in **nano** zu bearbeiten. Dabei ergab sich jedoch ein Problem: Nach dem Bearbeiten und vor dem Speichern der Datei bestand keine Möglichkeit, einen weiteren Check-In auszuführen, sodass die **Hashwerte** nicht verglichen werden konnten.

Alternativer Ansatz

Als Lösungsidee wurde eine temporäre Datei (**tmp-Datei**) eingeführt, mit folgendem Ablauf:

1. Nach dem Bearbeiten speichert der Nutzer die Änderungen in der tmp-Datei.
2. Der Hashwert der Originaldatei wird erneut berechnet und mit dem ursprünglichen Hashwert verglichen.
3. **Ergebnis des Vergleichs:**
 - Wurde die Originaldatei verändert, wird die tmp-Datei verworfen.
 - Andernfalls bleibt die Originaldatei unverändert.

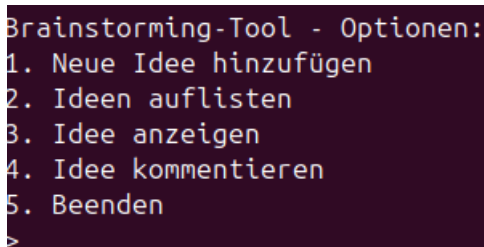
Offene Frage

Bei dieser Lösung wurde der **Rollback** nicht genutzt. Dies führte zu der Überlegung, ob das **Verwerfen der tmp-Datei** genauso sicher ist wie die Durchführung eines Rollbacks. Der Rollback hätte theoretisch nachträglich implementiert werden können, doch war unklar, ob dies tatsächlich erforderlich oder sinnvoll wäre. Diese Idee wurde schließlich verworfen.

Teil 2: Prototyp einer Brainstorming-Anwendung

Beschreibung

Im Rahmen der zweiten Aufgabe wurde ein **Brainstorming-Tool** entwickelt, das die **ZFS_Library** verwendet. Sämtliche Operationen wie **Lesen** und **Schreiben** wurden mithilfe der Bibliothek implementiert. Das Tool bietet eine benutzerfreundliche Bedienung über das Terminal.



```
Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
>
```

Funktionsweise

Das Brainstorming-Tool umfasst folgende Funktionen:

1. **Neue Ideen hinzufügen:** Der Nutzer gibt einen Titel (Dateiname) sowie eine Beschreibung (Text) ein.
2. **Titel auflisten:** Alle verfügbaren Titel werden angezeigt.
3. **Inhalt einer Idee anzeigen:** Der Inhalt einer spezifischen Idee wird angezeigt.
4. **Inhalt überschreiben:** Der Nutzer kann den Inhalt einer bestehenden Idee bearbeiten.
5. **Programm beenden:** Das Programm wird geschlossen.

Funktionen 2 und 3 sind nur lesend und daher nicht relevant für die Konfliktbehandlung. Konflikte können jedoch bei den Operationen 1 und 4 auftreten:

- **Operation 4:** Konflikte bei der Überschreibung von Inhalten wurden bereits in Aufgabe 1 beschrieben.
- **Operation 1:** Konflikte können auftreten, wenn zwei Nutzer zeitgleich dieselbe Idee erstellen oder eine Idee erstellt wird, die bereits existiert.

Fehlerbehandlung

Sollte der Nutzer versuchen, eine Datei zu erstellen, die bereits existiert, wird die folgende Fehlerbehandlung durchgeführt:

```
Brainstorming-Tool - Optionen:
1. Neue Idee hinzufügen
2. Ideen auflisten
3. Idee anzeigen
4. Idee kommentieren
5. Beenden
> 1
Titel der Idee (keine Leerzeichen erlaubt): Test
⚠️Die Idee existiert bereits. Möchtest du sie kommentieren? (ja/nein)
ja
Check-In abgeschlossen. Aktueller Hash: 904f7f42b7877bda9df5541677cdeede5d0e4fdad4f8e16c4b2f841eccc523d7
```

1. Der Nutzer wird informiert, dass die Datei bereits existiert.
2. Der Nutzer wird gefragt, ob die Datei überschrieben werden soll.
 - **Ja:** Der Nutzer wird direkt zur Bearbeitung der Datei weitergeleitet (Operation 4).
 - **Nein:** Der Erstellvorgang wird abgebrochen.

Diese Fehlerbehandlung gewährleistet eine nahtlose Interaktion und verhindert unbeabsichtigte Überschreibungen.

Teil 3: Validierung

Für die dritte Aufgabe wurden die Befehle der **ZFS_Library** mithilfe von Threads in schneller Abfolge ausgeführt, um gezielt **Konflikte** zu provozieren. Dadurch konnte im größeren Maßstab getestet werden, was in Aufgabe 2 bereits manuell geprüft wurde.

Erkenntnisse aus den Tests

Eine interessante neue Erkenntnis ergab sich im Zusammenhang mit der Verwendung von Snapshots: Wie bereits in **Teil 1** beschrieben, kann es vorkommen, dass das Laden des neuesten Snapshots zu lange dauert und währenddessen ein neuer Snapshot erstellt wird. Dank einer integrierten **Sperre** in ZFS wird jedoch verhindert, dass dabei Daten verloren gehen.

Testdetails

- **Setup:** Es wurden **10 Threads** erstellt, die jeweils **50 Operationen** ausführten.
- **Simulation der Nutzereingabe:** Um das Tippen eines Nutzers realistisch zu simulieren, wurde zwischen den beiden Check-Ins eine Pause von **200 Millisekunden** eingebaut.
- **Testparameter:** Der Text wurde direkt als Parameter übergeben, um den Bearbeitungsvorgang zu steuern.

Am Ende gibt das Programm eine kleine Auswertung aus. Die Summe aus Konflikten und Snapshots muss identisch sein zu überschriebenen Dateien und neuen Dateien. In diesem Fall 309 jeweils.

```
==== Auswertung ====
Dateien gelesen: 191
Dateien überschrieben: 282
Neue Dateien erstellt: 27
Gesamtoperationen: 500
Dauer der Ausführung: 53478 ms
Konflikte erkannt: 102
Snapshots erstellt: 207
```