Einleitung

Dieses Projekt implementiert eine Java-Bibliothek für ein ZFS-basiertes Dateisystem unter macOS, das Konsistenz durch die Nutzung von ZFS-Snapshots gewährleistet. Das System besteht aus mehreren Komponenten, die über das Terminal aufgerufen und bedient werden.

Die Hauptbestandteile sind:

- **ZFS_FS**: ZFS Dateisystem Bibliothek verwaltet ZFS-spezifische Funktionen, einschließlich der Initialisierung und der Verwaltung von Snapshots.
- **Client**: Implementiert das Interface Runnable und wartet nach dem Start in einer Endlosschleife auf Benutzerbefehle.
- **InitializeFS** (Main-Klasse): Fordert den Benutzer zur Eingabe eines Dateisystemnamens auf und ruft anschließend die Initialisierungsfunktion aus ZFS_FS auf.
- **ClientStarter** (Main-Klasse): Fordert den Benutzer zur Eingabe eines Dateisystemnamens auf und startet dann einen Client, der mit diesem Dateisystem arbeitet.

Die Transaktionslogik des Clients basiert auf einem Zeitstempelvergleich: Beim Lesezugriff auf eine Datei speichert der Client deren letzten Änderungszeitpunkt. Beim Commit wird überprüft, ob sich dieser Zeitstempel verändert hat. Falls eine externe Änderung vorliegt, erkennt das System einen Konflikt und setzt den Zustand auf den letzten Snapshot zurück.

Die Verwaltung der ZFS-Snapshots sowie die Initialisierung des Dateisystems erfolgen über ProcessBuilder innerhalb von ZFS_FS. Das Projekt setzt eine funktionierende Homebrew-Installation von zfs und zpool voraus. Da sowohl das Erstellen eines ZFS-Pools als auch dessen Verwaltung Administratorrechte erfordern, muss das Systempasswort eingegeben werden. Alle Versuche, diese wiederholte Passworteingabe zu umgehen, führten zu unerwartetem Verhalten.

Ein automatisiertes Testsystem zur gezielten Erzeugung und Analyse von Konflikten ist daher nicht möglich. Zwar lassen sich Clients direkt als Objekte instanziieren und deren Methoden

programmgesteuert aufrufen, jedoch verhindert die notwendige Passwortabfrage eine reibungslose Automatisierung. Das Projekt liegt unter https://github.com/mmaywald05/OS3.git und beinhaltet in der ReadMe eine Anleitung zum Aufsetzen des Projektes und zum Cleanup. Der manuelle Cleanup hinterher ist sehr wichtig. Es wird eine virtuelle Partition angelegt, auf der ZFS das Dateisystem aufsetzt. Nach dem man mit dem Programm fertig ist muss man die Partitionen manuell identifizieren und auswerfen und die ZFS Datenbanken löschen.

ZFS_FS

Initialisierung

Die Initialisierung des Dateisystems erfolgt über die main-Methode der Klasse InitializeFs.java. Voraussetzung dafür sind die Programme zfs und zpool, die unter macOS über Homebrew installiert werden können. Die entsprechende OpenZFS-Distribution ist unter Homebrew OpenZFS verfügbar.

Bei der Ausführung fordert das Programm den Benutzer auf, einen Namen für das Dateisystem (fsName) festzulegen. Dieser Name muss systemweit einzigartig sein, da er als Identifier in der ZFS-Datenbank verwendet wird. Anschließend führt das Programm die folgenden Schritte aus, um eine virtuelle Festplatte zu erstellen und das ZFS-Dateisystem darauf einzurichten:

1. **Erstellung einer 1 MB großen virtuellen Festplattendatei:**
   `truncate -s 1M ~/Desktop/fsName.img`
2. **Einbinden der virtuellen Festplatte ohne automatische Zuordnung eines Dateisystems:**
   `hdiutil attach -nomount ~/Desktop/fsName.img`
   die ID der erzeugten virtuellen Festplatte wird hierbei gespeichert.
3. **Erstellen eines ZFS-Pools auf der virtuellen Festplatte:**
   `sudo zpool create fsName disk`
4. **Setzen des Mountpoints für das ZFS-Dateisystem:**
   Um die Verfügbarkeit zu verbessern, wird das Dateisystem innerhalb des Projekts in einem dedizierten Ordner mountpoint bereitgestellt:
   `sudo zfs set mountpoint=projectRoot/mountpoint/fsName fsName`
5. **Zuweisung von Benutzerrechten:**
   Damit der Nutzer uneingeschränkten Zugriff auf das Dateisystem erhält, werden die Besitzrechte angepasst:
   `sudo chown $USER projectRoot/mountpoint/fsName fsName`

Nach Abschluss dieser Schritte ist das Dateisystem unter mountpoint/ verfügbar. Für jedes initialisierte Dateisystem wird dort ein gleichnamiger Ordner angelegt, der alle enthaltenen Dateien verwaltet. Änderungen können auch über den Finder vorgenommen werden, wodurch sich gezielt Schreibkonflikte auslösen lassen.

Weitere ZFS_FS Methoden:

Methoden für die Verwaltung von Snapshots; sind mit ProcessBuilder implementiert.

- createSnapshot()
- deleteSnapshot()
- rollbackSnapshot()

Generelle Utility:

- run(ProcessBuilder process): Führt process aus.
- run_output(ProcessBuilder process): Führt process aus und gibt Konsolenoutput als String zurück.
- run_admin(String cliInput, String prompt): Fragt Nutzerpasswort ab und führt dann den CLI-Input mit Administratorrechten aus.

Client

Bei Start des Clients wird der Name des Dateisystems abgefragt, für das der Client arbeiten soll. Die Client-Anwendung ist alt Thread implementiert, der in dauerschleife auf den Nutzerinput über die CLI wartet. Folgende Befehle werden unterstützt:

Command	Argument1	Argument2	Function
ls	-	-	Liste aller Ideen (Dateien in ZFS Filesystem)
create	[ideaName]	-	Erzeugt Datei mit Namen [ideaName].txt
read	[ideaName]	-	Schreibt Inhalt der Datei [ideaName].txt auf Konsole
write	[ideaName]	„[content]“	Überschreibt Inhalt der Datei [ideaName].txt mit [content]
append	[ideaName]	„[content]“	Hängt [content] an Inhalt der Datei [ideaName].txt an
help	-	-	Schreibt liste der Befehle auf Konsole
exit	-	-	Beendet die Anwendung

Beispiele

```
> create newIdea
File created: /Users/maywald/IdeaProjects/ZFS_lib/src/mountpoint/demo1/newIdea.txt
--------------------
```

Beispiel 3 Erstellen einer neuen Datei

```
> read newIdea
Upside-down Mirror?
--------------------
>
```

Beispiel 2 Lesen einer Datei

```
> write newIdea "Upside-down Mirror?"
Replacing idea newIdea with "Upside-down Mirror?"
Snapshot created: demo1@Client1
Commit? (y/n)
> y
No Conflicts. Committing transaction.
--------------------
```

Beispiel 1 Überschreiben einer Datei mit Conflict Ermittlung

Transaktionslogik

```
public void transaction(String filename, String content){
    createSnapshot();
    long t1 = getLastModified(filename);
    if(commitPrompt()) {
        long t2 = getLastModified(filename);
        if(t1 != t2){
            System.out.println("Conflict detected. Rolling back.");
            rollbackSnapshot();
        }else{
            System.out.println("No Conflicts. Committing
transaction.\n--------------------");

            write(filename, content);
```

```
            }
            deleteSnapshot();
        }else{
            System.err.println("Write to " + filename + " aborted.");
        }
    }
```

Es wird zuerst ein Snapshot genommen. Der commitPrompt wartet auf die (y/n)-Bestätigung des Nutzers über die Konsole. Damit kann man den Commit verzögern, um zwischenzeitlich einen Konflikt zu erzeugen. Die Konsistenzprüfung findet mit Zeitstempeln statt. Das schreiben von Dateien ist die einzige Operation, die eine Konsistenzprüfung erfordert.

Weitere Client Methoden:

- run(): Runnable Interface Method. Endlosschleife, die auf CLI-Input wartet.
- Snapshots
  - Create, delete, rollback
  - Legen den Namen des Snapshots fest „poolName@clientID“ und rufen entsprechende Funktion in ZFS_FS auf.
- Dateimanipulation:
  - create, delete, read, write, append
- Utility:
  - boolean commitPrompt: Warte, bis User “y” oder “n” eingibt. Gibt true zurück, falls y.
  - long getLastModified: Gibt Zeitstempel der letzten Änderung einer Datei zurück.

Beispiel für zwei Transaktionen mit Konflikt

Im folgenden sind die Outputs zweier Clients abgebildet, mit denen ich gleichzeitig auf demselben Dateisystem gearbeitet habe. Leere Eingaben „>“ repräsentieren Pausen. Die Chronologie der Befehle von oben nach unten wurde über beide Prozesse hinweg so gut wie möglich eingehalten.

```
maywald@Mac src % java ClientRunner Client1
Enter the Pool name:
demo1
Client Client1 waiting for input. Type 'help' for list of commands
> ls
idea.txt

> read idea
Put Jelly on both sides of toast.
------------------
> write idea "Client 1 changed this."
Replacing idea idea with "Client 1 changed this."
Snapshot created: demo1@Client1
Commit? (y/n)
> y
No Conflicts. Committing transaction.
------------------
>
>
>
>
> write idea "Client 1 is provoking a conflict with this edit"
Replacing idea idea with "Client 1 is provoking a conflict with this edit"
Snapshot created: demo1@Client1
Commit? (y/n)
> y
No Conflicts. Committing transaction.
------------------
> read idea
Client 1 is provoking a conflict with this edit
------------------
>
>
> read idea
Client 1 changed this.
------------------
> exit
Client Client1 exiting.
maywald@Mac src % #
```

```
maywald@Mac src % java ClientRunner Client2
Enter the Pool name:
demo1
Client Client2 waiting for input. Type 'help' for list of commands
>
>
>
>
>
>
>
>
>
>
>
>
> read idea
Client 1 changed this.
------------------
> write idea "Client 2 is changing this."
Replacing idea idea with "Client 2 is changing this."
Snapshot created: demo1@Client2
Commit? (y/n)
>
Commit? (y/n)
>
Commit? (y/n)
>
Commit? (y/n)
>
Commit? (y/n)
> y
Conflict detected. Rolling back.
> read idea
Client 1 changed this.
------------------
> exit
Client Client2 exiting.
maywald@Mac src %
```

// Warte auf Client1

// Warte auf Client1

Zur besseren Anschaulichkeit sind die Aktionen, die beide Clients ausführen, hier nochmal tabellarisch chronologisch dargestellt.

t	Client1	Client2	Bemerkung
1	Verbinde mit demo1	Verbinde mit demo1	‚demo1' = Name des ZFS Pools
2	Liste inhalt auf	Warte auf Input	
3	Lese Datei „idea"	Warte auf Input	
4	Überschreibe Datei „idea"	Warte auf Input	
5	Warte auf Bestätigung	Warte auf Input	
6	Führe Transaktion durch	Warte auf Input	Kein Konflikt
7	Warte auf Input	Lese Datei "idea"	Client 2 liest von Client 1 geschriebenen Inhalt
8	Überschreibe Datei „idea"	Überschreibe datei „idea"	Gleichzeitige Änderung
9	Warte auf Bestätigung	Warte auf Bestätigung	Client 2 braucht länger für die Bestägigung
10	Führe Transaktion durch	Warte auf Bestätigung	
11	Lese Datei „idea"	Warte auf Bestätigung	Client 1 liest eigene Änderung in Datei
12	Warte auf Input	Führe Transaktion durch	
13	Warte auf Input	Konflikt! Rollback	
14	Lese Datei „idea"	Lese Datei „idea"	Client 1 und Client 2 lesen wiederhergestellten inhalt