Übung 3

Repository: https://github.com/Raphael90K/BSA3

1 Hardware, Dateisystem und Hinweise

Zur Bearbeitung dieses Übungsblattes wurde ein Raspberry-Pi 5[1] verwendet. Dieser läuft unter dem Betriebssystem Ubuntu Desktop 24.04.2 LTS. Während der Bearbeitung des Übungsblattes wurden zwei unterschiedliche Ansätze getestet.

1. Eine 5 GB große Partition wurde auf einem USB Stick erstellt und als ZFS Dateisystem initialisiert und verwendet.
2. Ein 1 GB großes Loop-Device wurde initialisiert und als ZFS Dateisystem verwendet.

Da festgestellt wurde, dass die USB Partition bei größerer Anzahl an Snapshots und insbesondere beim bereinigen der Snapshots deutliche Ladezeiten aufwies, wurde im späteren Verlauf ausschließlich das Loop-Device verwendet. Dieses wurde als Verzeichnis `/zfs` eingebunden. In den nachfolgenden Kapiteln werden zwei Ansätze zur Lösung des Übungsblattes eingeführt, implementiert und validiert. Die zur Verwendung grundlegenden Shell Befehle sind auf der o.g. Github Website beschrieben. Bei der Lösung dieses Arbeitsblattes wurde bei der Codeerstellung ChatGPT[2] in der kostenlosen Version verwendet, u.a. um bestimmte zusammenhängende Funktionen von Java zu implementieren, die c Programme zu erstellen und die Brainstorming Anwendung auf Ebene des Dateisystems mit einer GUI aufzubauen.

2 Modellierung

Zur Bearbeitung dieses Übungsblattes werden zwei Lösungsansätze verwendet, die die Konsistenz der vorhandenen Dateien prüfen und eine Transaktionsverarbeitung bei Dateiänderung gewährleisten sollen.

[1] https://www.raspberrypi.com/products/raspberry-pi-5/
[2] https://chatgpt.com/

1. Die Überwachung erfolgt auf Ebene des Programms mit Hilfe des implementierten Frameworks. Transaktionen werden im Programmablauf implementiert.
2. Die Überwachung erfolgt auf Ebene des Dateisystems durch ein gesondertes Programm, sodass die Änderung an Dateien unabhängig vom Editor erfolgen kann.

Die unterschiedlichen Modelle werden in den folgenden Abschnitten beschrieben.

2.1 Überwachung auf Ebene des Programms

Diese Überwachung innerhalb des Programms erscheint auf den ersten Blick trivial und benötigt auch bei optimistischer Nebenläufigkeit keine Snapshots und Rollbacks. In der Theorie sollte es für mehrere Programme möglich sein eine Datei gleichzeitig zu öffnen und dann zu schreiben. Wenn das Programm eine Datei öffnet, wird der Hashwert der Datei im Programm gespeichert. Wenn die Datei gespeichert werden soll wird geprüft, ob der Hashwert der Datei unverändert ist. Falls ja, wird gespeichert und die Transaktion so beendet. Falls nein, wird die Datei nicht gespeichert.
Bei der Modellierung einer Transaktion auf Ebene des ausführenden Programms wird beim Öffnen der Datei eine Transaktion mit `start` begonnen und das Speichern mit einem `commit` beendet. Hierbei erfolgen folgende Schritte:

1. Bevor eine Datei geöffnet wird, wird eine Transaktion gestartet und ein Snapshot vom aktuell gültigen Zustand erstellt.
2. Der aktuelle Hashwert der Datei wird berechnet.
3. Die Datei wird geöffnet, kann gelesen und ein neuer Inhalt vorbereitet werden.
4. Wenn die Transaktion abgeschlossen werden soll, wird die zu schreibende Datei gesperrt.
5. Der Hashwert der Datei wird erneut berechnet.
6. Die Datei wird geschrieben.
7. Der Hashwert der Datei im letzten Snapshot wird berechnet.
8. Es wird geprüft, ob alle drei Hashwerte gleich sind.

   (a) Falls ja: Der Commit ist erfolgreich und ein Snapshot vom aktuell gültigen Systemzustand wird erzeugt.

   (b) Falls nein: Es gab eine Änderung außerhalb des Programms, mit gültigem oder ohne Snapshot. Ein Rollback auf den letzten gültigen Snapshot wird durchgeführt.

9. Die Datei wird freigegeben.

Es wird hierbei davon ausgegangen, dass mehrere Programme eine Transaktion gleichzeitig starten können. Das Programm, das die Transaktion als erstes beendet, ändert die Datei konsistent im Vergleich zur geöffneten Datei und erstellt einen gültigen Snapshot. Die nachfolgenden Änderungen führen zu Inkonsistenzen, sodass ein Rollback zum Snapshot der ersten Änderung erfolgt. Durch das Sperren der Datei für den kurzen Zeitraum des `commit` soll sichergestellt werden, dass die Datei während die Konsistenz geprüft und die Änderung vollzogen wird nicht von anderen Prozessen geändert werden kann.
Änderungen, die beispielsweise in einem Texteditor durchgeführt werden, führen ebenfalls zu einem Rollback, jedoch bleibt die Datei bis zum nächsten `commit` als gültig bestehen. Aus diesem Grund wird die in Abschnitt 2.2 beschriebene Überwachung, unabhängig von lesenden und schreibenden Programmen, ebenfalls modelliert.

2.2 Überwachung auf Ebene des Dateisystems

Im zweiten Ansatz soll das Dateisystem selbst, respektive das Wurzelverzeichnis des ZFS Dateisystems von einem Programm überwacht werden. Hierbei soll in erster Linie überwacht werden, wenn eine Datei geöffnet oder geändert wird. Unter der Berücksichtigung der zuvor genannten Annahme, welche Transaktionen Gültigkeit besitzen, muss das Programm folgende Schritte ausführen:

1. Beim Start des Programms wird ein initialer Snapshot erstellt.
2. Wenn ein Programm eine Datei öffnet wird ein Snapshot erstellt und der aktuell gültige Hashwert der Datei berechnet.

3. Wenn eine Datei geändert wird, wird der Hashwert im letzten Snapshot berechnet und es wird geprüft, ob beide Hashwerte gleich sind.
   (a) Wenn ja: Die Transaktion ist gültig und ein Snapshot wird erstellt.
   (b) Wenn nein: Die Transaktion ist inkonsistent und es erfolgt ein Rollback zum letzten konsistenten Zustand.

Mit diesem Ansatz sollten sich auch Änderungen außerhalb eines bestimmten Programms überwachen lassen. Ein Nachteil ist jedoch, dass Dateien für den Abschluss einer Transaktion nicht sinnvoll gesperrt werden können. Dies kann bei hochfrequenten Änderungen einen Nachteil für die Atomizität einer Transaktion bedeuten.

3 Implementierung

Für beide Ansätze wurden Java Programme erstellt. Diese werden in den nachfolgenden Abschnitten beschrieben.

3.1 Überwachung auf Ebene des Programms

Die notwendigen Java Dateien befinden sich im Package `zfsmanager`. Für die Transaktion ist die Klasse `TransactionManager` zuständig. Diese stellt die grundlegenden Methoden `start` und `commit` bereit. Mit `start` wird ein Snapshot erzeugt und die Transaktion gestartet. Bei Aufruf der Methode `commit` wird die Datei gesperrt, die Konsistenz geprüft und die Transaktion mit einem Snapshot oder einem Rollback abgeschlossen. Die Klasse `ZFSFileManager` stellt die benötigten Methoden zum Erstellen der Snapshots und zum Rollback bereit.

3.2 Überwachung auf Ebene des Dateisystems

Die Java Dateien zur Überwachung des Dateisystems befinden sich ich im Package `zfsFileMonitor`. Zudem werden zwei c Programme im Ordner `c` benötigt. Zunächst war vorgesehen die Überwachung ausschließlich in Java zu realisieren. Hierzu stellt Java eine Klasse

`WatchService` bereit, die verwendet werden kann, um verschiedene Dateioperationen zu überwachen [baPi24]. Ein Problem ist jedoch, dass hier nur das Ereignis und die modifizierte Datei festgestellt werden kann. Eine Möglichkeit zu überwachen wer, oder welches Programm die Modifikation vornimmt, wurde nicht gefunden. Aus diesem Grund wurde `fanotify` verwendet. Hierbei handelt es sich um eine Linux API zur Überwachung des Dateisystems, welche verschiedene Events unterstützt und zusätzlich die PID des Ursprungsprozesses wiedergibt [man724]. Das Programm `c/fanotify.c` überwacht `open` und `modify` Events auf einem angegebenen Verzeichnis und schreibt diese in eine `fifo`[3] Pipeline, die dann vom Java Programm verarbeitet werden kann. Die Überwachung der PID ist notwendig, da ansonsten nicht zugeordnet werden kann welcher Prozess eine Datei öffnet oder modifiziert.

Der Einstieg in das Java Programm bildet die Datei `ZFSFileMonitorPID.java`. Diese nimmt die Events der Pipeline entgegen und verarbeitet sie in der Methode `handleFileChange`. Während der Verarbeitung wird `fanotify.c` über ein Semaphore gesperrt und anschließend wieder freigegeben. Der Grund hierfür ist, dass das Berechnen eines Hashwertes ein neues `open` Event auslöst, was dann zu einer Endlosschleife führen würde. Nach der Freigabe wird der Puffer von `fanotify.c` durch ein `read` geleert und die Überwachung fortgesetzt. Hierdurch kommt es in der Zeit des Methodenaufrufs `handleFileChange` dazu, dass keine Überwachung des Dateisystems stattfindet.

Nach und nach werden für die verwendeten Dateien Einträge in einer `Map` angelegt, die den PIDs Hashwerte zuordnen, die beim Öffnen der Dateien berechnet werden. Zudem werden die letzten Snapshots ebenfalls gespeichert. Die Transaktion erfolgt dann nach folgendem Schema:

1. Bei einem `open` Event wird der aktuelle Hashwert der Datei berechnet und der Datei und PID zugeordnet. Anschließend wird ein Snapshot erzeugt.
2. Wenn das Programm die Datei anschließend speichert wird ein `modify` Event erzeugt und der vorher berechnet Hash der PID mit dem Hashwert im letzten Snapshot verglichen.
   (a) Wenn beide gleich sind, ist die Transaktion gültig.
   (b) Wenn beide ungleich sind, gab es in der Zwischenzeit eine erfolgreiche Änderung und

[3] https://man7.org/linux/man-pages/man7/fifo.7.html

die Transaktion ist ungültig. Ein Rollback wird durchgeführt.

3. Zum Schluss wird erneut ein Snapshot vom Systemzustand erzeugt.

Hashwerte und Snapshots werden in einem `RingBuffer` verwaltet, da festgestellt wurde, dass das beim Ändern einer Datei in einem Texteditor unmittelbar vor dem Speichern ein erneutes Öffnen stattfindet. Hierdurch wird ein neuer Hashwert und Snapshot erzeugt, was natürlich bei einer zwischenzeitlichen Änderung zu keiner Inkonsistenz bei der Erkennung führt. Bei unmittelbar aufeinanderfolgenden Dateiänderungen wäre so ein weiteres Zurückspringen möglich. Die Funktionen der Klasse `Timer` sollen kurze Prozesszeiten berücksichtigen und bei zwei unmittelbar aufeinanderfolgenden Ereignissen einen weiter zurückliegenden Snapshot und Hashwert verwenden. Dies hat jedoch bis zuletzt nicht immer zuverlässig funktioniert. Ein Grund war mitunter, dass die Dateiüberwachung für den Zeitraum der Verarbeitung ausgesetzt wurde und eventuell wichtige Events nicht verarbeitet werden. Bei der Implementierung der Brainstorming App wurde daher durch die Implementierung ein erneutes Öffnen der Datei unmittelbar vor dem Speichern unterbunden.

4 Brainstorming Anwendung

Die Brainstorming Anwendung für die Überwachung auf Ebene des Programms ist in der Datei `tfsmanager/Brainstorming` als Konsolenanwendung implementiert. Diese bietet die Möglichkeiten die Liste der Ideen anzuzeigen, Ideen hinzuzufügen, zu kommentieren und zu löschen. Beim Kommentieren wird eine bestehende Datei ausgelesen und ein neuer Text angehangen. Das Hinzufügen, kommentieren und bearbeiten wird jeweils von einem `start` und einem `commit` umgeben, um die Gültigkeit der Transaktion zu prüfen.
Die Brainstorming Anwendung für die Überwachung auf Ebene des Dateisystems wird ebenfalls in der Konsole verwendet. Die optische Darstellung erfolgt jedoch mit Hilfe des Framworks `lanterna`[4]. Dies ermöglicht die gleichen Funktionalitäten wie die zuvor genannte Implementierung, wobei die

[4] https://github.com/mabe02/lanterna

Transaktionsüberwachung auf Ebene des Dateisystems erfolgt. Hierzu müssen zunächst die Programme `fanotify.c` und `ZFSFileMonitorPID` gestartet werden. Abbildung 1 zeigt das Hauptmenü der Anwendung.

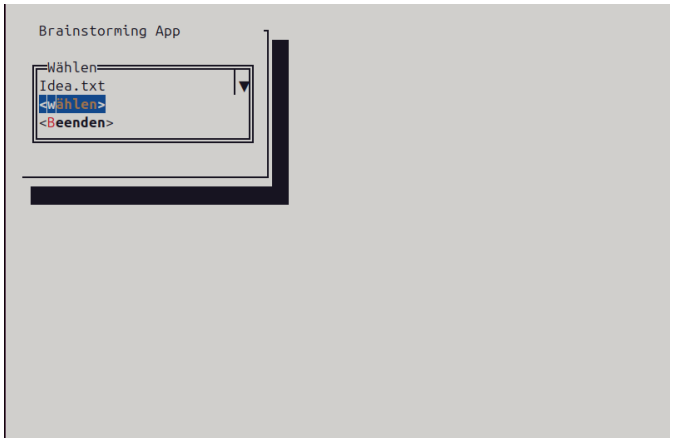

Abbildung 1: Hauptmenü der BrainstromingApp

Über das Dropdown Feld kann der Nutzer durch die angelegten Ideen wechseln, eine neue Datei anlegen oder eine bestehende Datei löschen. Die gewählte Aktion wird mit dem `wählen` Button ausgewählt. Abbildung 2 zeigt das Bearbeitungsfenster für eine geöffneten Idee. Diese kann flexibel bearbeitet und kommentiert werden.

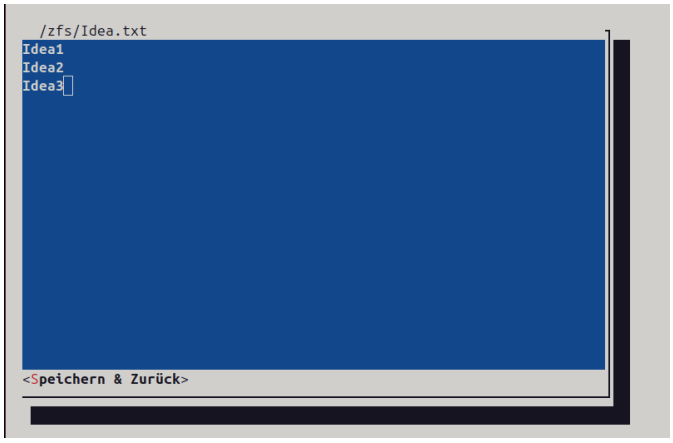

Abbildung 2: Texteditor der BrainstormingApp

Diese Brainstroming App befindet sich im Package `brainstorming`.

5 Validierung

Für beide Implementierungsansätze wurden unterschiedliche Validierungen implementiert. Diese werden in den folgenden Abschnitten erläutert und ausgewertet.

5.1 Auf Ebene des Programms

Das automatische Validierungstool für das Framework, welches die Konsistenz auf Ebene des Programms testet, ist im Verzeichnis `testA` abgelegt. In diesem greifen eine festgelegte Anzahl an Threads auf zehn vorher angelegte Dateien zu. Hierbei öffnen sie zufällig Dateien, warten eine zufällige Zeit [0, 499] und schreiben anschließend einen zufälligen Buchstaben in die Datei. Ein Durchgang dauert 10 Sekunden und die Anzahl an zugreifenden Threads wird von 2 bis 16 erhöht (* 2). Nach Beendigung werden die gültigen Commits, Rollbacks und Fehler dokumentiert. 32 Threads waren aufgrund der eingeschränkten Leistung des verwendeten Raspberry Pi nicht mehr möglich. Die Ergebnisse sind in Abbildung 3 dargestellt.

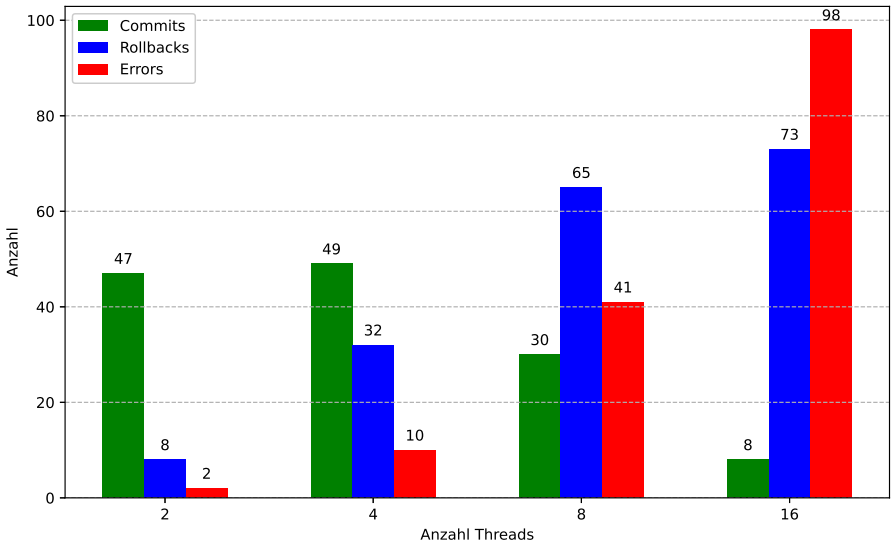

Abbildung 3: Anzahl Ergebnisse der Transaktionen in Abhängigkeit der Anzahl der Threads

Zu erkennen ist, dass der Durchsatz an gültigen Commits mit steigender Anzahl möglicher zeit-

gleicher Zugriffe deutlich abnimmt und zunächst die Anzahl an Rollbacks steigt. Je Höher die Wahrscheinlichkeit von Kollisionen ist, desto höher steigt auch die Anzahl der Fehler. Hierbei handelt es sich um `OverlappingFileLockExceptions`, die auftreten, wenn sich die Wahrscheinlichkeit von Zugriffsversuchen während eine Datei gesperrt ist erhöht. Bei den gemessenen Werten ergibt sich ein Druchsatz von maximal 5 erfolgreichen Transaktionen pro Sekunde.

5.2 Auf Ebene des Dateisystems

Aufgrund der ausgesetzten Überwachung erscheint ein ähnlicher Test wie zuvor nicht sinnvoll, da eine Menge an Events während ein Event verarbeitet wird unberücksichtigt bleiben. Aus diesem Grund wird die Zeit untersucht, bei der das System noch stabil überwacht werden kann. Hierzu öffnen und schreiben zwei Prozesse eine Datei nach folgendem Schema:

1. Prozess A öffnet die Datei.
2. Prozess B öffnet die Datei.
3. Prozess B schreibt die Datei. Datei wurde bisher nicht geändert, daher konsistent.
4. Prozess A schreibt die Datei. Datei wurde geändert, was zu einem Rollback führt.

Der Ablauf ist in Abbildung 5 dargestellt. Dieser Vorgang wird jeweils 20 Mal wiederholt. Hierdurch dürften in die Datei nur der Buchstabe B geschrieben und jeweils 20 Rollbacks durchgeführt werden. Die Konsolenausgabe für einen erfolgreichen Durchgang ist in Abbildung 5 dargestellt. Die Programme befinden sich im Package `testB`.

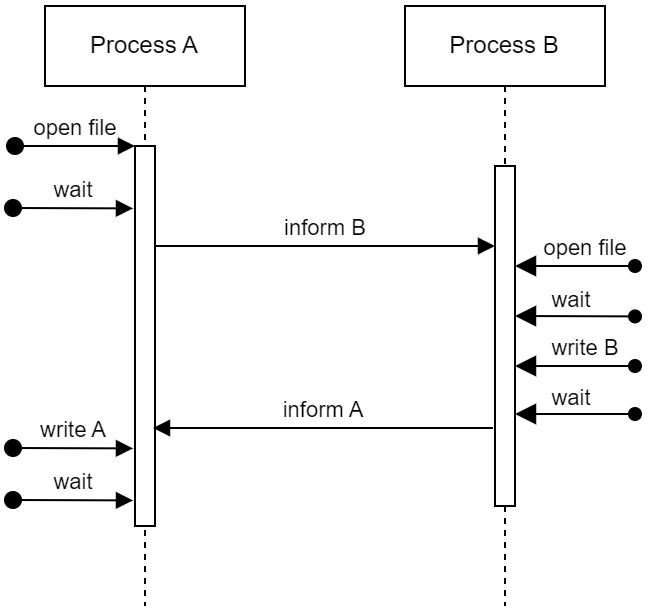

Abbildung 4: TestB

Zwischen den unterschiedlichen Events warten die Prozesse jeweils eine gewisse Zeit. Diese Zeit wurde auf 5, 2 und 1 Sekunde gesetzt und die Programme jeweils ausgeführt. Hierbei stellte sich heraus, dass bei Wartezeiten zwischen 5 und 2 Sekunden alle Events erfolgreich überwacht und alle 20 Rollbacks an den richtigen Stellen durchgeführt wurden. Bei einer Wartezeit von einer Sekunde wurden lediglich 9 Rollbacks durchgeführt, wodurch es zu Inkonsistenzen bei der Transaktionsverarbeitung kam. Die geschriebenen Dateien befinden sich im Ordner **results**.

6 Fazit

In diesem Übungsblatt wurden zwei Vorgehensweisen bei der Überwachung von Transaktionen auf dem einem ZFS Dateisystem implementiert. Die erste Implementierung überwacht die Transaktionen auf Ebene der ausführenden Programme, was eine Integration der Transaktionslogik in den Programmablauf bedeutet. Bei diesem Ansatz kann es jedoch weiterhin zu Inkonsistenzen kommen, wenn ein anderes Programm eine Datei ändert und gerade keine Transaktion läuft oder während der Zeit nach dem Start und vor dem Commit einer Anwendung. Auch kommt die Implementierung nicht ohne ein Filelock aus, welches während des Commits die Datei sperrt. Hierdurch dürfte

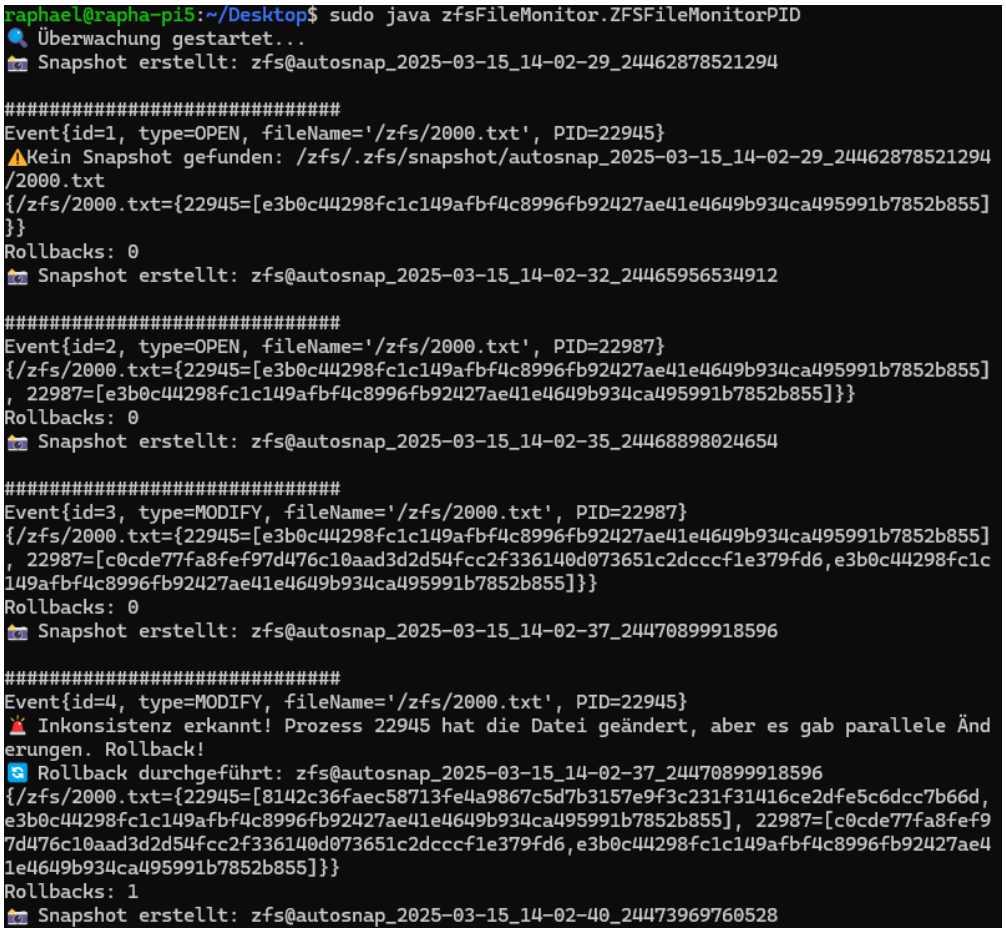

Abbildung 5: Überwachung auf Ebene des Dateisystems

aber die Atomizität der Transaktion gewährleistet sein.

Die zweite Implementierung zur Überwachung des Dateisystems ist deutlich flexibler und soll auch Änderungen unabhängig vom ändernden Programm erkennen und zurückrollen können. Da das Programm jedoch keinen direkten Einfluss auf das Lesen und Schreiben hat, ist die Atomizität nicht gewährleistet, wenn Programme hochfrequent auf die gleiche Datei zugreifen. Aufgrund von Prozesszeiten, in denen keine Überwachung stattfindet, können Events übersehen werden. In den Tests auf einer einfachen Hardware konnte jedoch ermittelt werden, dass Events erfolgreich überwacht werden können, wenn zwei Sekunden zwischen ihnen liegt. Dies erscheint für einen alltäglichen Gebrauch theoretisch ausreichend. Sinnvolle Verbesserungen wären eine Lückenlose Überwachung der Events und die Behandlung von erneuten `open` gefolgt von `modify` Operationen,

die unmittelbar zusammenhängen.

In diesem Übungsblatt fand eine tiefere Auseinandersetzung mit den Möglichkeiten und Grenzen des ZFS Dateisystems statt. Zudem konnte die Interprozesskommunikation, welche bereits Gegenstand des zweiten Übungsblattes war, bspw. durch die Verwendung von Shared Memory und Semaphores wiederholt werden.

Literatur

[baPi24] baeldung und G. Piwowarek. A Guide to WatchService in Java. https://www.baeldung.com/java-nio2-watchservice, 2024. Stand: 08.01.2024, Aufruf am: 17.03.2025.

[man724] man7.org. fanotify(7) — Linux manual page. https://man7.org/linux/man-pages/man7/fanotify.7.html, 2024. Stand: 17.11.2024, Aufruf am: 17.03.2025.