1456880
Späth, Maximilian

Operating Systems Exercise 3

Optimistic concurrency using ZFS snapshots

Accompanying repository at: GitHub_OperatingSystemsLecture3_Späth

General Information

As I am running Windows on my Host systems, I went to install Virtual Box and the Ubuntu Desktop LTS as VM. I gave a main disk of an arbitrary amount of 60gb for the operating system and two secondary disks of 20GB to try out the ZFS mirror capability. Further the VM has 4 cores and 8GB of RAM.

Before starting with the exercise, I installed prerequisites such as JAVA 21 and ZFS. I made sure that ZFS works properly by setting up the two secondary disks as a mirror and experimented with it. For programming I chose to stick with my IntellijIDEA on my Host system. To easily allow the VM to access the project, I added the project as shared folder to the VM which is a functionality provided by Virtual Box. By this I could easily program on my host system, save changes to the files and evaluate right in my VM compiling and running the classes. After this setup was completed and I tested ZFS functionality as well as the programming environment I started to think about the exercise.

Basic Concept

To enable **optimistic concurrency for file modifications** using ZFS snapshots, the program needs the ability to perform basic commands on the bash, e.g. ZFS commands like create snapshot, rollback to snapshot or file commands like append to file, create file and some logic to detect conflicts.

The basic approach that I follow here is to **simulate file writing through sleeping threads**, that, when it attempts to write on a file, **tells a central instance about its action**, simulates a user editing of the file by sleeping and then **commits (user saving) his changes by telling the central instance** that he has finished. The **central instance** is therefore **responsible for detecting if any conflicts occur** before it commits the changes of the thread to the file. **If it detects a conflict, it calls the rollback** with the snapshot taken on the start of the threads file editing. A **conflict** is given if the **timestamp of the file** that the thread wants to commit to is **different to the timestamp the central instance saved** alongside the created snapshot when the thread started editing the file.

A thing I learned while implementing this is that ZFS prohibits the behavior of rolling back to a snapshot when newer snapshots exist, e.g. while two threads were unknowingly running towards a conflict, other threads may have started working on different files, leading to newer snapshots. ZFS has some functionality that determines which of these newer snapshots should be deleted before rolling back to an older snapshot. However, if you still want to roll back to this older snapshot, you need to delete the newer ones beforehand which ZFS does for you if you provide a certain flag within the rollback command. As we want to model the behavior though of strictly rolling back to the snapshot taken before file editing on conflicts, we delete newer snapshots utilizing the flag. This means that threads that were written on the files for which the snapshots got deleted through such a rollback are affected by the rollback as well running into a conflict. This is also checked by the central instance, that changes are only committed if

1. The snapshot still exists
2. No conflict, e.g. changed timestamp is detected

If none of these options are determined by the central instance, then the changes are made.

Implementation

The functionality that achieves the concept described is handled within the class **ZFSMapper.java**. Throughout the **Java ProcessBuilder** was used to execute commands on the Linux System. The class has a method *doCommand(String command)* that accepts a command and executes the command using the ProcessBuilder. Based on this, simple methods with no functionality besides calling doCommand with the corresponding command were written:

DEBUG:

- *public static void showFileContent(String fileName)*
- *public static void removeFile(String fileName)*
- *public static void showFiles()*
- *public static void showSnapshots()*

DEBUG/PreInit:

- public static void deleteAllSnapshot()

FUNCTIONALITY:

- *public static synchronized void deleteSnapshot(String nameOfSnapshot)*
- *public static void createSnapshot(String nameOfSnapshot)*
- *public static void createFileWithContent(String fileName, String fileContent)*

The methods implementing conflict logic are:

- *public static void **rollbackToSnapshot**(TransactionInformation transactionInformation)*
- *public static synchronized TransactionInformation **notifyWrite**(String threadName, String filename)*
- *public static synchronized int **appendToFile**(TransactionInformation transactionInformation, String content)*
- *private static long **getLastModified**(String fileName)*

The datatype **TransactionInformation** is a helper class holding the information of a transaction e.g.:

- SnapshotName
- FileName
- ThreadName
- LastModified

Another Key structure is the

HashMap <String,TransactionInformations> transaction

held within the **ZFSMapper**. It basically acts as a **mirror to the current active ZFS snapshots and maps the timestamp of the file edit start to the snapshot corresponding to that event**. The key to is the snapshot name which is built like 'threadName-FileName'.

Explanation Conflict Functionality Methods

In the following I explain the key methods implementing conflict functionality.

`public static synchronized TransactionInformation notifyWrite(String threadName, String fileName`

When a thread starts its simulated writing to a file, it calls these methods on the ZFSMapper. Without hesitation due to our optimistic transaction, the ZFS Mapper creates a transaction information object Z for this transaction, saving the last Modified timestamp, and the snapshot name for which a snapshot is created accordingly. It puts Z in the transactions and returns Z for the thread that is called these methods such that it can pass Z later on when calling appendToFile. As threads concurrently call this method it is important it is set to synchronized.

`public static synchronized int appendToFile(TransactionInformation transactionInformation, String content)`

When a thread awakes from its sleeping e.g. the user has finished editing and saves his changes, he calls this method, passing its transaction Information object created by notifyWrite together with the content it wants to commit. Following the concept, the ZFSMapper checks the two conflicts that can occur before applying any changes to a file.

Case1, no snapshot existing:

For case 1 we test if the transaction map contains the key e.g. the snapshot name held within the transaction information object. If the key is not existing, this means that the snapshot was affected by a rollback to another snapshot which is handled by ZFS. We comply with this functionality and count this as a conflict in which the changes are not committed to the file. We return code 1 and end here.

Case2, timestamp inequality:

Comparing the last Modified timestamp of the file we want to commit to, and the timestamp saved on transaction initialization results in inequality means another process altered the file e.g. a yet unhandled conflict has occurred. In this case we need to roll back to the state the snapshot captured on transaction initialization discarding any changes made to the file since then. We call rollbackToSnapshot which handles further functionality and return 2 if the method is done.

Case3, no conflicts:

If the above conflicts did not occur, we can commit the changes the thread made and call the bash command. We further delete the snapshot as it's no longer needed thereby calling the ZFS destroy command as well as removing its representation from the transactions map. This may seem counterintuitive at first but effectively it does not make any sense to keep the snapshot. It has solved its purpose as backup, if an error occurs on this file, the changes of this thread may be revoked anyway and the snapshot used for this was made by another thread. Further this thread may start working again on this file which will lead to the snapshot names being no unique identifiers anymore. However, as the corresponding snapshot is not needed anymore after successfully appending content, it can be safely deleted, and the snapshot names will remain unique identifiers.

As this method is also subject to concurrency it needs to be synchronized as well.

`public static void rollbackToSnapshot(TransactionInformation transactionInformation)`

Being called only from within the synchronized appendToFile method, this method is not subject to concurrency and does not need the synchronized keyword. It is only called when a rollback should be carried out and was detected due to a thread committing his transaction information object. As we take a lot of snapshots, it almost always will happen that we roll back to an older snapshot and newer snapshot exists. ZFS default behavior is to prohibit this. We use the -r flag to force the rollback leading ZFS to delete all newer snapshots. This is the side effect leading to case 1 in appendToFile as this makes currently working threads running into conflicts and not committing which is okay as we cannot ensure the file is still the same as it was when they began writing.

As our currentTransactions snapshot did its job it is no longer needed and deleted. Further the snapshot mirror map transactions are updated with the remaining snapshots which effectively should always be none, as we kinda start from new.

`private static long getLastModified(String fileName)`

retrieves and return the lastModified timestamp of the file.

1456880
Späth, Maximilian

Brainstorming Tool:

When I started conceptualizing this tool, I thought of one main bash in which the current accessible files were listed as options and an option where you could create a file. The options are selectable by the number in front of the option. While creating File would be in the main bash, prompt for file title and idea content, I wanted to spawn a new bash for when you enter a number in de main prompt corresponding to an existing file, e.g. opening that file to append some text. However, I ran into several problems on my way to implementing this with terminal managers/emulators like xterm, gnome-terminal or tmux. The key problem here would be the managing of the multiple processes for which some work would have been done for inter-process communication which would be doable but requires some effort. My problems though were more basic, like for example starting bashes from within my running program and then being able to write input to them. Although this was able when I manually used the command line, running it by program was not possible after having spent a few hours on this issue. Following the exercise description I therefore settled on a simpler approach, in which I would start a transaction internally, call a text editor on the same file and if changes are made in the text editor and saved, the main thread that started editing on its own, opened then the text editor and waited for it to close, would encounter an conflict and rollback.

As I already implemented file option and file creation logic in the bash for my earliest attempt on an actual brainstorming tool, starting the BrainstormingTool.java will prompt with the options file creation and all files in the directory. If none exists, you create one following the simple procedure and afterwards are prompted again to select and option.

Here you select the file, a text editor on this file is opened and a thread is already waiting in the background with its own file editing on this file. If you make any changes and save, a rollback is carried out by the thread waiting in the background, e.g. main Thread. The main program is held in a while(true) loop, and with normal usage no errors should occur so you can chain the different combinations of writing-not saving, writing -saving, and see how changes are rolled back or how the other threads write to the file if you choose to not edit anything.

1456880
Späth, Maximilian

Validator:

To test the conflict resolving efficiency of the implemented system I keep following the methodology of simulating user editing of files by sleeping threads. The Validator therefore simulates multiple users by thread that randomly access files and append content to them all based on the ZFSMapper. If a conflict occurs, then a rollback follows deleting all newer snapshots resulting in threads that will not append their content to a file and start on the next. There are 5 parameters:

- NumberOfThreads
- IterationsPerThread e.g. attempts to write to a file
- NumberOfFiles
- meanWritingTime e.g. mean of Gaussian distribution
- StDWritingTime e.g. std of Gaussian distribution

The gaussian distribution is used to compute the writing time a user spends on a file e.g. the sleeping time of the thread.

Before conducting an experiment, I tried out values during my programming and after I found the concept to be well implemented and functioning. Obviously if file size and number of threads are close together it is way more likely for conflicts to occur, as well is it if the mean and std are both high as then some thread will spend a lot of time on a file while other threads may be hopping and are more likely to land on the file the long thread is writing to. Further in a environment where rollbacks occur often and new file writings occur often, a lot of transaction will not finish as there snapshots were already deleted.

The results are captured in the following values:

- transactionsAttempted e.g. numberOfThreads * iterationsPerThread
- successFullWrites
- conflicts
- rollbacks
- meanRollbackTime in ms
- conflictRate e.g. conflicts/transactionsAttempted

The matter with successFullWrites is, that there is no functionality implemented to track back the successful writes that were affected by a rollback and I can't think of a quick and even dirty way to do that. Therefore, this value is way higher than the actual value if many rollbacks occurred.

As a conflict are counted Attempts to write to a file but the snapshot was deleted by some rollback and attempts to write to a file but the timestamp is unequal. Rollbacks are obvious, so is the mean time for which a nested Timer class exists in the ZFSMapper using System.nanoTime. ConflictRate is again obvious.

1456880
Späth, Maximilian

Experiment:

For the experiment I set on 10 simulations e.g. 10 different Parameter settings. The runs can generally be distinguished in two groups, 1-4 and 5-10.

In group one I've set the number of threads to three while the number of files was 9 for all four runs. Further the writing time distributions were varied in this group.

For group two, I wanted to test how the rollback times increase with high numbers of files and threads. To have more comparability I used a thread/file ratio of 1/10 and 1/50 where a pair of 1/10 and 1/50 ratio would have the same number of threads but the corresponding number of files.

In the following I first discuss the results of the runs 1-4 and afterwars the results of 5-10.

Run	1	2	3	4	5	6	7	8	9	10
Parameters										
iterPerThread	150	150	150	150	150	150	150	150	150	150
#threads	3	3	3	3	10	10	30	30	50	50
#files	9	9	9	9	100	500	300	1500	500	2500
meanWriting	2000	2000	600	200	600	600	600	600	600	600
stDWriting	200	1800	400	100	400	400	400	400	400	400
Results										
TA_Attempted	450	450	450	450	1500	1500	4500	4500	7500	7500
#Success	286	326	315	302	962	1349	2692	3475	4193	5547
#Conflicts	164	124	135	148	538	151	1808	1025	3307	1953
#Rollbacks	57	45	49	57	63	20	108	52	130	63
Mean_Rollback	340.49	322.11	319.21	312.16	372.50	383.03	564.28	633.74	824.94	915.75
Conflict_Rate	36.44	27.56	30.00	32.89	35.87	10.07	40.18	22.78	44.09	26.04

Runs 1-4:

The results suggest that the different writing times did not have a big impact. This can be seen at the mean_rollback time in ms, which settles around 330ms. As the rollback times is mainly influenced by the pointers ZFS needs to reset to the old copies, it is not highly influenced by the number of changes but rather by the number of files which, for these runs where all the same. For different numbers of files like for example 4 I found in manual testing a similar rollback time of around 300ms so I think it is like a lower bound on the system I tested that the ZFS rollback might take at least 250-300ms. For the number of rollbacks that occurred I find 45-57 relatively high for 450 writing attempts, but as the thread/file ratio for a free thread is 2/9 it is quite likely a thread that begins writing lands on a file where some thread Is already writing especially as they instantly begin writing on a file if they are finished. I can't come up with an explanation of why there are 12 less rollbacks in parameter configuration 2 than in parameter configuration 1 other than RNG related. It overall seems like having a low variance for the gaussian distribution leads to more conflicts e.g. rollbacks, but having runs 1 and 4 compared to 2 and 3 is not enough evidence for any such claims.

However, to test the impact on ZFS, I conducted the second experiment with a higher amount of thread and files, to stress test the conflict detection and rollback mechanic. I increased the number of files to up to 2500 files for the last run, as the time a rollback takes mainly relies on the complexity of the data hierarchy and is not related to the contents inside the files. This is due to the CopyOnWrite(CoW) mechanism which enables rollbacks to reallocate pointers for restoring old data. As I don't have deeply nested data hierarchy with lots of files, I force intense rollbacks using high amounts of files and therefore need more threads to ensure conflicts. Further I used the same parameters for the gaussian distribution of writing times as I did not want to make any further assumptions here nor did I want them to have an impact. It may further be noted that the number of files is in the same ratio as the amount of threads, e.g. 1/10 and 1/50. In the following the results of Runs 5-10 are presented:

Runs 5-10:

Regarding the conflict ratio the thread/file ratio is represented in the results, with the 1/10 ratio having conflict rates of 35% in run 5, 40% in run 7 and 44% in run 9, while the 1/50 ratios have a very low 10% for the run 6 and a little higher 23% in run 8 and 26% in run 10. Comparing these results to the conflict rats of the 1/3 ratio in the first runs I cannot find a pattern indicating that the high amount of threads and files induce different behavior and might be maybe as well RNG related to how the threads access the files.

What can directly be seen though is that the mean rollback time increased from run-to-run showing that the increased parameters have an impact on the rollback time. The highest rollback time is found in the last run at 915 ms, while the lowest is found at 372 ms in the first (5.) run. However, that every successive run has a higher rollback time indicates that the rollback time is not only influenced by the amount of files, as the runs 7 (564ms) and 9 (824ms) have higher rollback times as the runs 6 (383ms) and 8(633ms) although they have fewer files (7= 300, 9 = 500) (6= 500, 8= 1500). What does strictly increase as well and probably lead to the reason for the rollback to take longer is the amount of successful writes. Although I mentioned they appear higher than their real value, because I cannot track back what successful writes are reverted, this increasing number over the runs is still a strong indicator for why the rollback time increases as well. As the thread/file ratio mainly influences how many successful writes can be carried out, and less rollbacks occur, more files will be changed once a rollback occurs. I therefore assume that the amount of time a rollback takes is surely on the one hand based on the number of files existing but secondly and mostly on the number of files that were actually subject to a change in the first place. Only by this can I explain the outcome of the experiment, that the time a rollback takes increases from run to run although the number of files does not. I therefore am sure that the key factors I can observe here for the time a rollback takes are the amount of changed files for which he has to adjust the pointers. Because if he would simply revert all pointers to the old copies, then the highest rollback times would be achieved by the highest file numbers which is not the case.