Dept. of Eng. PC & Informatics Polytechnic School, University of Ioannina

Spring semester 2023
Teacher: S. Anastasiadis

MYY601 Operating Systems

Announcement: Thursday 9th March, Delivery: Friday 31st March at 21:00

Lab 1: Implementing multithreading in a data storage engine

1. Introduction

You are given a storage engine implemented in the C language. You are asked to implement the multi-threaded operation of the put and get commands provided by the storage engine. Your implementation will allow multiple threads to call the put and get commands simultaneously. The storage engine should perform concurrent operations correctly and maintain statistics of the execution time of each operation. Your implementation should be in C and based on the Linux Pthreads library.

1.1 LSM tree-based storage engine

The source code given to you implements the storage engine**Kiwi**based on log-structured merge tree (LSM-tree). Storage engines are an important part of modern cloud infrastructures as they are responsible for storing and retrieving data on a machine's local devices. A distributed storage system uses thousands of such machines to achieve scalable and reliable operation (eg, Facebook, Google).

The LSM-tree is the data structure that storage engines often rely on. The provided API includes put and get functions for key-value pairs. The put function accepts as a parameter a key-value pair to be added to the structure. The get function accepts a key as a parameter and requests the corresponding value if there is a key-value pair stored in the structure with the given key, otherwise it returns an error if the key is not found.

For quick lookup, the structure keeps the stored data sorted in memory or disk. Typically, the most recent data is kept in memory in an ordered structure called a memtable, usually implemented with a skip list. A skip list is a data structure consisting of multiple lists organized at different levels to speed up the search for an element. When the memtable is full, it is moved to disk in order to free up memory and be able to accept new data. While the memtable remains in memory, there is a log file that receives incoming data and caches it on disk for quick recovery in the event of an error.

Files stored on disk are organized into multiple levels whose size increases geometrically from the lowest to the highest levels (0..6). A disk file in this structure is called a Sorted String Table (sst) because the data is sorted by keys containing strings. When the memtable is moved to disk, it is merged with the files (at level 0 or 1) whose keys overlap with the memtable's keys. Additionally, file compaction is triggered when the number of files in level 0 exceeds a predefined threshold (eg, 4), or the total size of files in a level exceeds a predefined threshold for that level. With collapsing, one level's files are merged into one file that is added to the next level.

1.2 Multi-threaded implementation

In the implementation given to you, there are already two threads on the storage engine. The first thread calls a sequence of put or get operations one after the other as defined by the command line, while the second thread is responsible for compressing files when the required conditions mentioned above are met.

The put function interacts with the memtable to insert a new element into the structure. If required, the current memtable is merged with an sst file, which may result in further collapsing from one level to the next.

The get function searches for the supplied key first in the memtable and if not found there in the sst files starting at level 0 and going to subsequent levels as needed. It is possible that a memtable has been merged, so it should also be searched before the sst files. The files to be searched are selected based on the range of keys they contain, assuming that it is unnecessary to search an sst whose key range does not contain the key being searched for. Searching an sst file is made faster with a Bloom filter that contains the hashes of the keys already entered in that file.

Given the complexity of the code, you are asked to plan the implementation in steps. A trivial implementation would use a lock to implement mutual blocking on put and get operations. An improvement would allow multiple readers or one writer to work at a time. Additional steps would use different locks for the memtable and sst files so that different threads can run concurrently as long as they don't modify the same part of the system at the same time. The solution should take into account that there is already a thread responsible for the merges that asynchronously modify the sst files in the background.

You are responsible for documenting the changes you make to the code both with comments in the source code you submit and with a detailed description in the report you submit. Code that does not make sense and is not adequately documented will not be considered for grading, or may negatively affect grading.

1.3 Performance Statistics

The implementation given to you includes some very basic performance metrics for read (get) and write (put/add) operations. You are requested to ensure that performance metrics are correctly obtained on the multi-threaded implementation that you submit. Proper multi-threading means that the performance recording variables are updated individually by the different threads using locks for mutual exclusion.

Your report should include performance tests in different scenarios that you deem appropriate. Examples are workloads that contain only put or only get, as well as a mix of put and get according to specific percentages for each operation. You can present the results either simply with a screenshot or better with graphical representations (eg, from excel).

2. Preparation

Download the virtual machineMYY601Lab1.zip (1.2GB) and extract it to your local hard drive. The virtual machine runs version 10 (“Buster”) of Debian Linux 64-bit. Download and install itVMware Player v15 on your machine. The machine is set to use 1 CPU core and 1GB of RAM, but this is easily changed from the virtual machine settings. To take advantage of the windowed environment, ensure that you are running the virtual machine in full-screen mode by clicking the appropriate icon at the top left of VMware Player. You can login as a simple user with the name myy601 and the same as the password. If you need to install additional packages on the virtual machine's Linux system, you can login as root (with password myy601) and run**apt install <package>**. The virtual machine has the C programming language and the Firefox browser installed. For your convenience, you can copy-paste text and files between virtual operating systems machine and your machine's operating system, e.g. to port the code you've written when submitting with turnin.

Brush up on your C knowledge and make sure you understand the Pthreads library function calls to create and synchronize threads. Additionally, familiarize yourself with the debugger**gdb**in order to step through and see the current contents of various variables (Cheatsheet &Manual). You will need to use basic virtual machine tools to open the source code files (eg, nedit, xemacs). Open a terminal and navigate to the directory**~/kiwi/kiwi-source**which contains the source code of the storage engine and a simple benchmark (**kiwi-bench**). Build the executables by running**make all**and scroll through the directory**bench**which contains thebenchmarks. You can input a number (eg, 100,000) of items into the machine with the command**./kiwi-bench write 100000** and then read them with the command**./kiwi-bench read 100000**.

3. Work

The task asks you to implement put and get operations that can be executed on different threads.

i. First you need a basic understanding of the path taken by put operations (**db_add**) and get (**db_get**). You don't need to understand all the details of the code but get a basic idea of how a function is served by either the memtable or the sst files. It is helpful to navigate through the code using the documentation available online at the following link (Code documentation).

   a. In the Data Structures option you will find the definition of important structures such as **_db**,**_memtable**,**_skiplist**,**_sst**. Especially in the structure **_sst**they already exist type synchronization variables**pthread_mutex**or **pthread_cond**where need to keep in mind when improving system concurrency.
   b. In the Files option you can find important header type files that define the interface of important sections of the code. In particular, you should familiarize yourself with the file**db.h**and the functions you will modify (B.C., **db_add**,**db_get**), and the corresponding functions in**memtable.h**, **skiplist.h**, and**sst.h**.
   c. The file**config.h**contains important definitions ofmacros that define system behavior (e.g.,**BACKGROUND_MERGE**).

ii. Run the benchmark**kiwi-bench**with different crowds of imported and wanted items and monitor the messages in the terminal. Run the code in gdb (**gdb kiwi-bench**) to track the functions executed during the different executions.

iii. You can start multiple threads in benchmark (with appropriate extension of **bench.c**) and call get functions that don't conflict with structures. But you can't easily do this with put operations because it will lead to a segmentation fault and possibly leave the structure in inconsistent form. If you find a problem with the saved data, you should delete the directory**testdb**with**make clean**or**rm -rf testdb**in the directory**bench**.

iv. Determine the level of concurrency you want to achieve. This depends on the parts of the structure involved in the various operations and whether they only read the data or additionally modify the data. A simple approach is to allow only one operation to run at a time and secure this with mutual blocking. You should improve this trivial solution by adding additional concurrency by using multiple critical regions and experimenting with reader-writer synchronization at the various levels of the LSM-tree (eg, skip list, sst, merge, collapse, etc).

v. You can expand the command line arguments of the file**bench.c**in order to specify the mix of put and get operations as well as the percentage of each type (eg, 50-50, 10-90, etc.).
vi. You should maintain performance statistics of operations (eg, response time, throughput) with the necessary synchronization for mutual exclusion between different threads updating their respective variables. At the end, you can print the statistics you measured.

Grading will be based on the functionality you've implemented, the quality of the code you've added, and the comprehensible documentation of it in the report you deliver.

4. What you will deliver

You can prepare the solution individually or in groups of up to three people. Submission will be made by one team member only. Submission after the deadline results in a deduction of 10% of the grade per day up to 50%. For example, if you submit your solution 1 hour after the deadline, the maximum score you can receive is 9 out of 10. If you submit your solution one week after the deadline, your maximum score drops to 5 out of 10. Submit the solution you in time with the order

```
/usr/local/bin/turnin lab1_23@myy601 Report.pdf kiwi-source.zip
```

You will be graded according to the description contained in the file**Report.pdf**and her accompanying implementation on file**kiwi-source.zip**. In the file kiwi-source.zip include only the files you have changed by specifying the path they are in in the source code you are given. Your report includes the names of the team members and your registration number. It also describes what you have implemented and how you did it by describing each line of code you added or modified. Lines of code that you changed or added but are not included in the report do not count toward the final grade, or may count negatively if they demonstrate fundamental deficiencies in your programming knowledge.

You are encouraged to add comments to the source code you wrote to make it easier for your examiner to understand your work. You should also include the output of the final command in your report**make**as well as the exit from the execution with various parameters. In the run output include only the command you used to run it**kiwi-bench**and the performance metrics without the detailed debug information that the run produces, unless there is something specific you want to show.**If the execution outputs included in the reference do not match the code you submit, the entire exercise will be zeroed out for obvious reasons.**

The file**kiwi-source.zip**you submit is a compressed file with themodified archives . For each file you changed you will only send the modified file and not the original file. You should perform**make clean**in the directory**kiwi-source**before creating the filezip to not include executable files (.o, .a, etc). The code you deliver should be able to be compiled with the tools included in the virtual machine you are given.

Bibliography

[1]https://github.com/google/leveldb