»What I cannot create, I do not understand«

– Richard Feymann

Performance Workshop Homework

Implement your very own Key-Value-Store in a programming language of your choice. While you can of course code it in Bash, SQL or even Brainfuck, for best learning effect, a compiled language is *recommended*. The internal design of your store can be whatever you like it to be as long it fulfills the requirements below. A basic LSM (Log-Structured-Merge-Tree), as discussed in the introductionary slides, is a good and performant choice though.

Deadline: You should prepare a basic prototype until the first workshop about CPU internals. After every workshop today you should have some ideas how to extend or improve your implementation. Try to see if you can incorparate some knowledge from the slides into your implementation:

https://sahib.github.io/misc/performance/slides/0_toc/index.html

Required minimal feature set

1. »`Get(key []byte) ([]byte, error)`« returns the value for `key` and possible errors.
2. »`Set(key, val []byte) error`« remembers `val` for `key` and returns possible errors.
3. When `Set(key, val)` returns, a subsequent `Get(key)` must immediately return the updated value.
4. The store can remember more values than there is physical memory.
5. The data must be stored persistently and loaded from disk on restart.

CPU Tasks:

1. Write benchmarks to measure performance of `Get()` & `Set()`
2. Profile your program using a profiler and identify bottlenecks.
3. Try to fix at least one of those bottlenecks.
4. Run your benchmarks again and see if it improved.

Memory Tasks:

1. Measure the number and amount of allocation.
2. See where the allocations come from and check if you can reduce them.
3. Measure again and repeat until you identified most allocations.
4. Bonus: Implement a sorted index to lower the number of keys you can keep in memory.

I/O & Syscalls Tasks:

1. What kind of syscall does your program use? Can you identify them all?
2. Can you reduce the number of syscalls or use more efficient ones?
3. Measure the write throughput and latency of your store on full load. (`Get()` & `Set()`)
4. Make your DB crash at random points and see if all data is written (`fsync()`)

Concurrency Tasks:

1. Provide an asynchronous API for your store so users do not block.
2. Implement (segment) compression, with background I/O.
3. Do the IO in background.
4. Queue up writes to the database.
5. Try to fetch keys in parallel.

Optional tasks for the motivated:

1. Implement segment compression
2. Implement »`Delete(key) error`« using tombstones.
3. Make sure `Get()` performs well if the key does *not* exist.
4. Implement transactions (write several values or none at all).
5. Implement »`Snapshot(w io.Writer) error`«, which streams a *consistent copy* of the database to `w` (which might be stdout or a file or a socket…). This can be used as backup.

6. Implement efficient range queries ($O(\log n)$) that can list all keys with a certain prefix (i.e. *Pat* matches *Patrick*, *Patricia*, *Pathological*, ...)