CS 744 Autumn 2020
PA4 - Building a Key Value Store

This assignment requires you to build a performant, scalable key-value server from scratch using C. The assignment can be done in a group of up to 3 students.

Multiple clients connect to the key-value server (**KVServer**) and once connected repeatedly send requests for either getting the value associated with a key or putting in a new key-value pair. Clients and servers will communicate using the given messaging format (**KVMessage**). Communication between the clients and the server will take place over the network through sockets. The server must use a cache (**KVCache**), which is backed by a persistent storage (**KVStore**) to store key value pairs since all keys cannot be stored in memory.

KVServer Design

KVServer will consist of a **main thread** that will perform the following steps:

1. It will read a config file and set up a listening socket on a port specified in the config file.
2. Create **n** threads at startup which will be specified in the config file.
3. Perform accept() on the listening socket and pass each established connection to one of the **n** threads in a round robin fashion. For example, the first connection is passed on to the first thread of those n thread for processing, and so on.

The KVServer will also have **worker threads** that are responsible for the following actions:

1. Each worker thread will set up and monitor an epoll context for the set of connections it is responsible for.
2. It runs an infinite loop that will monitor its client connections for work (in addition, if there are new connections for the thread, it must add it to the epoll context). After reading requests from clients, it will process those requests, and write a response back to the client.

Key Value Cache: The server will have limited memory (specified via a config file parameter that lays out how many key value entries the server is allowed to hold in memory) and so this will serve as cache to the files that will hold the persistent representation of the data. The cache will be a fully associative cache (flat structure) with two possible replacement/eviction policies:

a. LRU:
b. LFU: Least Frequently used

Dirty entries evicted from this cache will have to be written out to the persistent representation which will be a file. You are free to use locality to evict more than one entry and bring in additional entries by locality to drive performance.

Persistence: All keys and corresponding values are stored on disk in files. How you organize the keys into files is up to you. We suggest that you keep keys that are “close” to each other in a single file. You also need to think about how keys are arranged in a file - if you want to read in one key (and corresponding value), you should do a **`lseek`** to the specific place in the file where the key will be. The meta data about key positions and files that exist can be in memory - think about using bitmaps.

Interfaces:

- Your key-value server will support 3 interfaces:
  1. Value **GET** (Key k): Retrieves the key-value pair corresponding to the provided key.
  2. **PUT**(Key k, Value v): Inserts/Overwrites the key-value pair into the store.
  3. **DEL(**Key k): Removes the key-value pair corresponding to the provided key from the store.
- You must use the exact request/response formats defined later in the specification for communication between external and internal components.
- 'Keys' and 'values' are always strings with non-zero lengths. They cannot be nulls either.
- Each key and value cannot be greater than **256 bytes**. If the size is breached, return an error. (Assume each character to be 1 byte in size)
- When PUTing a key-value pair, if the key already exists, then the value is overwritten.
- When GETing a value, if the key does not exist, return an error message.

- You should ensure the following synchronization properties in your key-value service:
  1. Reads (GETs) and updates (PUTs and DELETEs) are atomic.
  2. An update consists of modifying a (key, value) entry in both the cache and persistent representation (you can choose to do this lazily).
  3. Do NOT lock the entire cache for each write. Only lock that entry so that operations on other keys can proceed in parallel. If you do use locks, use multiple readers single writer locks for efficiency.

KVClient

The client is a separate process that will first establish a connection with the server, and then send **GET**,**PUT**, and **DELETE** requests to the server process. It will have a main and use the client library interfaces to do the actual operations.

KVClientLibrary

This is a library, which will encode and decode your request and response messages.

For example, at the client side this library will encode your request message to the decided message format, and then send it out to the server process. Similarly, it will decode the response received from the server, and then hand out the decoded response to the KVClient module.
Something complimentary will happen at the server end.

KVMessageFormat

Each request and response will be 513 Byte messages. First byte of the message will be some status code, and this status code will uniquely identify different requests and responses. Next 256 Bytes will be key, you can use padding if the actual key is not 256B. The last 256 Bytes of the message will be value, again you can use padding if value is not 256B. In case of **GET** and **DEL** requests where no value is required you can only consider the first 257B of the request message.

Status Codes for request message

GET: 1
PUT: 2
DEL: 3

Status Codes for response message

Success: 200
Error: 400 (with the appropriate error message)

Reasons for error could be GET key not found, DEL key not found etc.

Performance Analysis

Ensure the parallel execution of the system as much as possible using threads and locks. Read locks should be sharable which means multiple GET requests should be served concurrently. Keep the locking granularity as a single entry in case of KVCache and a single file in case of KVStore.

You should do a performance analysis of your server by running multiple clients against it and benchmarking its throughput. Submit a graph of Throughput Vs load and Response Time (as seen from the client) vs Load (in requests per second).

Submission Guidelines

- Submit via a tarball on Moodle by exactly one team member. Name it **<rollnum1>-<rollnum2>-<rollnum3>pa4.tar.gz**
- Have a **README** file which explains how to **build AND run** the server and client code and also mention the name and roll no. of all team mates (on the top).
- Make sure to have a single Makefile to build all your C files and execute them.
- There will also be a viva/demonstration.