CS 214 Programming Assignment 4

Memory Cache Daemon (100 points)

In this assignment you will write a C server that acts as networked memory cache. A memory cache is used as a cache to store binary large objects (BLOBs) in memory. A Blob is just a byte array that is identified by a binary key.

Memory caches are typically used in large clusters of servers where the data sets are larger than a single machine can hold. The data is thus “sharded” (divided) across multiple machines in the cluster. For example, a web site might have a large image set, where each machine holds some fraction of the images on its local disks.

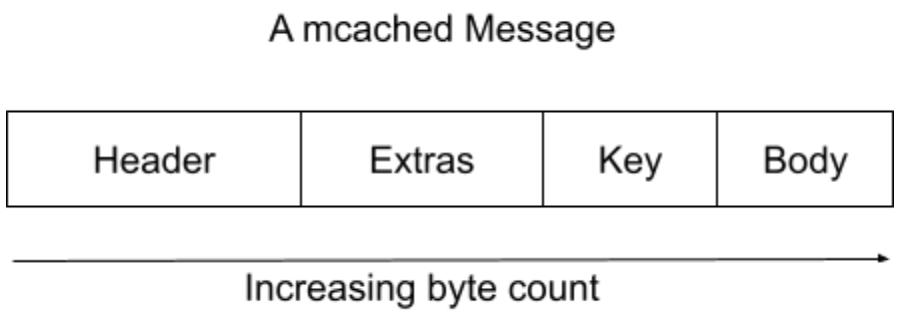
Reading from remote memory is typically faster than reading from hard drives, as accessing a machine’s memory over a local network can take several hundred nanoseconds compared to milliseconds for disk. This speed differential means a cluster environment can create one large cache that spans the memories of all the machines in the cluster, thus reducing the latency of retrieving requests. For example, a large web site may have many text pages and images. A cache of recent requests could be stored across the memories of multiple machines in the cluster. When an incoming web request enters the cluster, the object IDs (via a hash) map the hash to a specific server in the cluster. The request is then sent to the machine holding a copy of the object, which returns the object from memory rather than disk.

In this assignment, you will implement a single server called “mcached”, for memory cache daemon (recall in computing a daemon is a background process that runs continuously, waiting for requests that trigger actions). The server is at its core a hash table that stores multiple key/value pairs which are accessible by any client over the network. Clients can add, delete, and query the server’s hash table. The server should support multiple simultaneous requests, so should be multithreaded.

The message format between the client and server is based on the memcached protocol, as below.

The types inthe protocol are defined as unsigned integers of 8,16,32 or 64 bits in big-endian format. For example a `uint16_t` is an unsigned 16 bit integer in big-endian format. The types are defined in `<sys/types.h>`. Messages are passed between the client and server via a TCP socket on an IP address and port known to both the client and server.

A message in the protocol consists of a fixed length header, as below, followed by the extras, the key, and the value (data) of the entry. The first byte, the magic number, is always 0x81. The lengths of the extras, key, and value (body) are sent in the header (again, in big-endian format).



Request and Response Header Format:

```
typedef struct {
      uint8_t magic;   // a number (0x81) to identify a cached header
      uint8_t opcode;       // the command or response code
      uint16_t key_length;  // length of the key, in bytes
      uint8_t extras_length;   // additional expansion field (not used)
      uint8_t data_type;       // can ID the type of data (not used)
      uint16_t vbucket_id;      // virtual bucket number. The response
must match the vbucket_id of the request.
      uint32_t total_body_length;  // the length of the data, in bytes
      uint32_t  opaque;          // field used to match responses with
requests (not used)
      uint64_t cas;          // compare and set token (not used)
} __attribute__((packed)) memcache_req_header_t;
```

Overall Description of the mcached Server:

The mcached server is started with 2 arguments: a server port and a number of threads to handle requests.

The server must implement a method to store the keys and values in the messages, for example, in a hash table or linked list. Each entry must insure thread-safe access to that specific entry. That is, if one thread is adding a key/value pair value and the other is removing it, the server must ensure the set of key/value pairs are not corrupted by adding appropriate synchronization primitives (e.g. mutexes or semaphores).

One descriptive version of the server would built it as:

Parse the command-line arguments for the server port and the number of worker threads to create.

Create a server socket and bind it to the port specified on the command line.

Listen for incoming connections.

Create the specified number of worker threads. Each thread is given access to the server socket file descriptor.

Each worker thread runs a loop that continuously accepts new client connections on the server socket. For each accepted connection, the thread calls a function to process requests from that client. After the function finishes, the worker thread closes the client connection and goes back to waiting for the next connection.

The function reads data from a connected client, expecting requests formatted according to the binary protocol. It reads the request header, checks its validity, and then reads the remainder of the request body (containing the key and value, depending on the command). Note that the constants for the commands are included in the file “mcached.h”.

Based on the command opcode in the header, the function to handle a client performs the following actions:

GET: Calculates the hash of the key, searches the set of keys for a matching key. If found, it locks the entry, sends a response containing the value, and unlocks the entry. If not found, it sends a "Not Found" response.

SET/ADD: Calculates the hash of the key. It acquires a lock on the hash table. It searches for the key.

If the command is ADD and the key exists, it releases the lock and sends an "Exists" response.

If the key is not found (for SET or ADD), it allocates memory for a new entry, copies the key, initializes the entry's lock, and inserts it into the set of key/value pairs.

It then locks the individual entry (releasing the main lock if necessary), frees any existing value, allocates memory for the new value, copies the incoming value, sets the value length, and unlocks the entry. Finally, it sends a success response.

DELETE: Calculates the hash of the key. It acquires the main lock on the set of key/value pairs. The function searches for the key. If found, it locks the entry, removes it, unlocks and destroys the entry's mutex, frees the memory for the key, value, releases all locks, and sends a success response back to the client. If not found, it releases the lock and sends a "Not Found" response.

VERSION: Sends a response containing the server version string. The version string is: "C-Memcached 1.0”.

OUTPUT: Acquires a global lock on the hash table. Takes a single unix timestamp with `clock_gettime(CLOCK_REALTIME …)`. Next, iterate through all key/value pairs, printing the hexadecimal representation of the seconds, nanoseconds, and each key/value pair to the standard output, all in hexadecimal, separated by a “:”, one per line. That is,each key value pair

is printed as 4 hexadecimal tuples as <seconds>:<nanoseconds>:<key>:<value>. It then releases the global lock and sends a success response.

Any other opcode: Sends a generic error response.

Client to Server Commands:

```
#define CMD_GET  0x00
#define CMD_SET  0x01
#define CMD_ADD  0x02
#define CMD_DELETE  0x04
#define CMD_VERSION 0x0b
#define CMD_OUTPUT  0x0c
```

Server Responses:

```
#define RES_OK          0x0000
#define RES_NOT_FOUND   0x0001
#define RES_EXISTS      0x0002
#define RES_ERROR       0x0004
```

Server Includes:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "mcached.h"
```

How to get started:

Start with a simple TCP server from the example in class. Read 1 message from the client and generate a response, for example, from the ADD message. Iterate adding message types until the server can handle all messages as a single threaded program.

Once all message types have been completed, add pthreads and move the code that handles messages to the thread's work function.

Submission Requirements:

If you are working in a group, you **must submit as a group in Gradescope**. See the link below for instructions.

You must submit a zip file with all the files needed to build your server, including a makefile where the default rule builds a binary called “mcached” (all lowercase). The program should link against the pthread and rt libraries; i.e., add “-lpthread -lrt” to the rule building the executable.

How to add group members in gradescope: https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members

Grading:

Your program will be tested against 5 test cases in increasing order of difficulty. Each test case is worth 20 points for a total of 100 points. Two test cases, with a client of 1 and 2 threads, are included in the project directory.

Testcases involve a multithreaded client issuing a carefully chosen sequence of mcached commands that verifies that all operations work correctly. Since commands such as ADD and SET are not fully independent in terms of testing (ADD after a SET on the same key should return RES_EXISTS), the test cases don’t test each command independently, but as a list of linked commands.

Each testcase has the same number of client threads, list of commands, and data for the key/value pairs, but we will increment the number of server threads (your code) and verify that as the number of server threads goes up, the time taken for all requests to complete goes down. If your server doesn't have multithreading, but just queues connection requests from client threads, you will only get points for the one test case where server thread count is 1.