Project 2 - multi-threading and network connections

For this project you will write a simple networked database server, with a single network listener thread and multiple threads storing data in individual files. Your server will implement a database of up to 200 keys; each key has an associated value of up to 4096 bytes, and network requests can write (to new or existing keys), read, or delete the value associated with a key.

1. Overview

Your server will have 6 threads:

- The original main thread. This will listen on the console and implement two commands:
  - "stats", which prints statistics, and
  - "quit", which terminates the server.
- The listener thread, which will open a TCP socket, accept incoming connections, and add work items to a queue for the worker threads.
- Four worker threads, which will wait for new work items and handle requests to write, read, or delete individual database keys.

You are given a fairly detailed description of how this should be done; note that some of the details are marked as "suggestions", while the remaining ones are requirements. In addition, there are implementation hints, which you are under no obligation to follow but which may make your work easier.

You are given three files: `Makefile` , `proj2.h` , and `dbtest.c` . The header file defines the request structure for your database, and `dbtest.c` is testing program you can use to verify the operation of your code. You will need to create a fourth file named `dbserver.c` and add rules to the makefile to compile it. You may need to install a package using the following command in the terminal to compile dbtest.c.

```
$ sudo apt-get install libz-dev
```

Hint: Although your final deliverable needs to have 6 threads, I would suggest implementing it in a single thread first. In other words write the following functions:

1. `listener(void)` - creates and binds the listening socket; loops accepting connections and calling:
2. `handle_work(int sock_fd)` - this reads a request from a TCP connection and performs the requested action (write/read/delete).
3. then implement `queue_work(int sock_fd)` , which allocates a work item and puts it on a queue, and `int sock_id = get_work()` , which gets an item from the queue and frees the work record. Now modify the listener thread to call `queue_work` followed by `handle_work(get_work())` .
4. Finally split these into threads - first the listener thread (still calling `get_work` and `handle_work`), then finally split out the handler threads from the listener thread.

Requirement: In your submitted code, in between accepting a new connection for a write and writing the file to hold its data, you must call `usleep(random() % 10000)` , which will randomly sleep between 0 and 10 milliseconds. This will (likely) cause any race conditions to show up in testing.

You will be using pthreads and their synchronization primitives including mutex locks and condition variables.
Here is a simple tutorial on pthreads: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html.

Thoroughly read through the project description first before you start programming. Note that this document specifies the designs/requirements for the program in multiple sections, but the section order does not necessarily reflect the order that you should be implementing your program.

2. Listener and communication

Your program will listen on port 5000 for TCP connections.

Hint: it's not at all necessary, but may be helpful to have an optional command-line parameter to use a different port, since if your program crashes you may be locked out of using port 5000 for a minute or two. Google TIME_WAIT for more details.

```
    int port = 5000;
    int sock = socket(AF_INET, SOCK_STREAM, 0);                /* (A) */
    struct sockaddr_in addr = {.sin_family = AF_INET,
                               .sin_port = htons(port),      /* (B) */
                               .sin_addr.s_addr = 0};
    if (bind(sock, (struct sockaddr*)&addr, sizeof(addr)) < 0) /* (C) */
        perror("can't bind"), exit(1);
    if (listen(sock, 2) < 0)               /* (D) */
        perror("listen"), exit(1);

    while (1) {
        int fd = accept(sock, NULL, NULL);    /* (E) */
```

The steps are:

- A, create a "listening" TCP socket named `sock`, which will listen for incoming connections
- B, the address we'll bind it to. We need to convert (byteswap) the port number to network byte order, and we'll set the address to 0, which means we'll listen for port 5000 on any address on this machine
- C, bind the listening socket to port 5000
- D, tell the OS to start listening on it.
- E, block until we get a new connection. If we're slow about handling the connection and new ones come in before the next call to `accept`, the parameter "2" in the call to `listen` tells the OS to queue up to 2 incoming connections before telling additional connections that we're busy.

Note that I'm using the `perror` function to let me know what went wrong if either `bind` or `listen` fails; this way I'll know if the `bind` failed due to "address already in use", which means that I have to wait for the connection from my last (crashed) attempt to timeout before I can use port 5000 again.

You'll need the following include files:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Hint: You can find out what include files you need for a system call `syscall` or a library function `libfunc` with the commands `man 2 syscall` and `man 3 libfunc`. It's not always clear whether something is a system call or a library function, or that it even matters to those of us who aren't writing the OS, so try both 2 and 3. (those are "chapters" of the online manual pages; most commands you use from the shell are in section 1, section 8 has system administration commands, and 4-7 are various obscure things)

Note that sockets are file descriptors; although the listening socket is kind of weird, the per-connection sockets can be passed to `read` or `recv`, `write` or `send`, and `close`. For more information about networking and sockets, refer to: http://beej.us/guide/bgnet/html/.

3. Database commands and responses (messages being exchanged)

You'll receive commands in a 40-byte structure:

op (1)	name (31)	len (8)	[data - 'W' only]

The operation is a single byte: 'W' (write), 'R' (read), or 'D' (delete).

The name and length fields are text (up to 30 and 7 bytes), padded with zeros, so you can handle them as strings.

For a write command, this structure is followed by "len" bytes of data. For read and delete commands you can ignore the length field. Hint - you can parse the length field with `atoi`: `int len = atoi(req.len);`

You'll reply with the same structure:

status (1)	name (31)	len (8)	[data - 'R' only]

The status byte is either 'K' for success, or 'X' for failure.

The name field is ignored.

The `len` field indicates how many bytes of data follow the header in a read response, and is ignored in the response to a write or delete.

To handle an incoming connection you should:

1. issue a single read of size `sizeof(struct request)` bytes, reading the data into a `struct request`.
2. For a write, call `read` to read an additional `len` bytes from the connection into a separate buffer. (a 4096-byte buffer is guaranteed to hold any requests you'll be tested with)
3. Perform the operation, then write a header back with `status` equal to 'K' or 'X'.
4. For a successful read, set the length field in the header (before you write it...) and then write that many bytes of response data after the header.
5. Finally close the connection with `close`

Hint When you send and receive messages over TCP, it is important to read/recv exactly the same amount of bytes that the other network end write/send. For example, if a client send 100 bytes and the server reads 80 bytes by mistake, 20 bytes will still remain in the buffer. If a client sends another 100 bytes and the server reads 100 bytes this time, the first 20 bytes the server reads will correspond to the remaining 20 bytes from earlier client message, and the next 80 bytes will correspond to the first 80 bytes of the second 100 bytes sent by the client. If you are new to network and TCP programming, I recommend first using dbtest program to send one message (e.g., write to one key) and see if you can print out this message from your server program.

Hint It may help your testing to implement a "quit" command, with `op='Q'`; when you receive a quit command, close the listener socket and call `exit(0)` immediately, without writing anything back.

4. Database implementation

You'll store data in files named `data.0`, `data.1`, ... `data.199`. This (a) allows using database keys that contain illegal filename characters, like "/", and (b) requires you to lock a bunch of stuff instead of just letting the OS handle it.

In real life you'd use a map structure that someone else wrote, either as part of the language or a third-party library. I'm going to strongly suggest that you just use a simple table with linear search - you're guaranteed to have no more than 200 entries, so a 200-entry table with a name and status for each entry will work fine.

Hint Factor out the logic for searching for (a) a name in the table, and (b) a free entry. If you follow the hints above you'll get this working and tested in a single thread before going multi-threaded and needing locking - you may find yourself having to refactor your table logic when you add locks.

A database entry goes through the following states:

```
      +-------------------------<---+
      v                             | D
+---------+   W   +------+       +-------+
| invalid | ----> | busy | ----> | valid |
+---------+       +------+       +-------+
                     ^               |
                     |               | W
                     +-----------<---+
```

When a write request is received for a new key, that key (hint: and its associated index) will be in the "busy" state until you're done with `usleep` and writing the file - any attempt to read, delete, or rewrite it will result in an 'X' response. A write to a valid key will put it back into the busy state until the rewrite is complete, while a delete makes the key invalid. (preferably immediately - there's no requirement to call `usleep` on delete)

Hint You already need something in your table to indicate whether a sequence number is available or in use; just give it 3 different values.

Hint You might want to put your files in another directory, like `/tmp`, so they don't fill up your code directory.

Hint You can use the `sprintf` function to format a string using the same syntax as you would use with `printf`, e.g.:

```
int filename[32];
sprintf(filename, "/tmp/data.%d", sequence_number);
```

5. Work queue (distributing commands to worker threads)

Like project 0 you'll need a head and tail pointer, and an item structure with a "next" pointer and a field to hold the value. (in this case all you need to send is a single integer, the file descriptor)

Since you'll be using this structure in different threads it shouldn't be allocated on the stack - you should allocate it with `malloc` before putting it on the queue, and free it after retrieving it from the worker thread.

You'll need a mutex and a condition variable so that (a) you can update the queue safely, and (b) worker threads can wait until something has been added to the queue.

6. Storing / retrieving data from files

[suggestion] You can use the `read` and `write` system calls for file handling like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
 [...]

    int fd = open(filename, O_RDONLY);
    int size = read(fd, buf, sizeof(buf));
    printf("size %d\n", size);
    close(fd);
```

and this:

```
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if (fd < 0)
        perror("can't open"), exit(0);
    write(fd, buf, len);
    close(fd);
```

(`O_CREAT` means create the file if it doesn't exist, and `O_TRUNC` means truncate it (i.e. delete the contents) if it already exists)

Hint You may find it helpful to put this at the beginning of your `main` function:

```
    system("rm -f /tmp/data.*");
```

which will delete all the files from your previous program run. You might **not** want to delete them at the end of the program, since you may find yourself needing to look at them when things go wrong.

7. Command line (main thread's role after creating other threads)

The main thread should go into a loop reading from standard input, looking for two commands - "stats" and "quit". The standard pattern for reading lines is:

```
    char line[128]; /* or whatever */
    while (fgets(line, sizeof(line), stdin) != NULL) {
        ... do something with 'line'
    }
```

The resulting line will have a newline on the end, before the null character that terminates the string. You can either just look for `"stats\n"` and `"quit\n"`, or you can parse a word out of it:

```
    char word[8];
    sscanf(line, "%7s", word); /* max 7 chars+nul */
```

To test for string equality you can use `strcmp(string1, string2)` - it returns 0 if the two strings are equal. (and -1, +1 if string1 is less than or greater than string2, which is useful when you're writing a sort function, but confusing any other time.)

The stats command should print out the following:

- number of objects in the table

- the number of read, write, and delete requests received
- the number of requests queued waiting for worker threads
- the number of failed requests ('X' responses)

The `quit` command should close the listener socket and call `exit(0)`. You may also want to delete all your data files, although that's up to you.

8. Testing

You can test your implementation with the `dbtest` utility:

```
$ ./dbtest --help
Usage: dbtest [OPTION...]

  -D, --delete=KEY           delete KEY
  -G, --get=KEY              get value for KEY
  -m, --max=NUM              max number of keys (default 200)
  -n, --count=NUM            number of requests
  -p, --port=PORT            TCP port to connect to (default 5000)
  -q, --quit                 send QUIT command
  -S, --set=KEY VAL          set KEY to VALUE (e.g., ./dbtest -S key val)
  -t, --threads=NUM          number of threads
  -T, --test                 10 simultaneous requests
  -?, --help                 Give this help list
      --usage                Give a short usage message
```

You can use it in 3 different modes:

- individual requests, using `--set`, `--get`, and `--delete`. You'll use these to get your implementation up and running.
- load testing, with `--threads=N` and `--count=N` - this will issue a mix of write, read, and delete requests, and will verify the data returned.
- simultaneous requests with `--test`, which sends a mix of simultaneous requests in random order.

You will in general need two terminals for testing. You are expected to run both the server and the test programs on the same VM. For testing, log into your VM from two terminals and run the dbserver on one terminal and run the dbtest program on the other terminal.

9. Test deliverable (Bonus points: 3 pt. Counts only for this project)

You can write a series of scripted tests, in a file named `testing.sh` and include it in the submission.

Note that you can run commands in the background within a script (for automated testing we are running the server program in the background to avoid using multiple terminals):

```
# send quit command after 5 seconds, output FAILED if dbserver
# exits with exit(1). [exit(0) is OK, exit(1) is failure]
#
(sleep 5; echo quit) | ./dbserver 5001 || echo FAILED&

# give it a second to get up and running. A usleep command would be nice...
sleep 1
./dbtest --port=5001 --count=100 --threads=5
wait
```

(note that I'm specifying a port - this way I can use a separate port number for each test, and even if one of them crashes and leaves the port in TIME_WAIT state it won't break the rest of the tests)

Do not write this test script at the last moment when you are about to finish your implementation, but rather write this script as you implement the dbserver. Start from simple test cases (e.g., issuing 1 command and the printing db state to see if the command went through) that can be used even if you do not have a complete implementation of the dbserver. As you make progress add more heavy-weighted test cases in the script. Your final implementation should pass all the cumulated test cases that you wrote along the way.

10. General (but important) advice and debugging

Multi-threaded programs are **very difficult** to debug and **debugging can take forever**. Errors may not be reproducible, and your program might just hang if you misuse the locks. If your program crashes in the middle of an execution, this might be rather an easy case to debug which you could potentially pinpoint the cause with **gdb**.

Make sure to incrementally build and test your code little by little. Checkpoint your code frequently whenever you have built and tested a new feature; you will know up to which point your code works. This will not solve all the debugging issues, but it will greatly reduce your debugging time.

Like always **plan your time ahead and start working on the project early.**

11. What to submit

You should submit all relevant files to this project (excluding compiled object and executable files). Include a README.txt file which briefly describes the files that you have created/modified for this project. Place all files in a directory and zip the directory. Submit the zip file through the Canvas project page by **Mar 25, 11:59 pm**.

Expectations for your code:

- builds the program with a simple "make" command using the Makefile.
- no compiler warnings.
- no valgrind warnings for illegal accesses. (memory leaks are OK)