Multithreaded client and server

Context
Multiple clients making requests to a single server program.

How to run the program

1. Run the command `gcc -o server -pthread my_server.cpp`.
2. Run `./server n`, where n is the number of worker threads in the thread pool.
3. In a separate terminal, run `gcc -o client -pthread my_client.cpp`.
4. Run `./client`.
5. Input the total number of user requests throughout the simulation followed by the description of each user request.

CLIENT PROGRAM

1. The `struct client_req` has the following data members

```
int id;
int t; //the time at which the request has been made
char command[CMAX]; //request/the command issued
pthread_t client_thread_id;
pthread_mutex_t client_mutex;
// a mutex accquired whenever we are accessing/modifying
//data members of this stucture that can be changed by different threads
simultaneously
int client_socket_fd; // file descriptor of the socket associated with the
client
```

2. The command/request issued can be in one of the following formats :-

- insert *key value*
- delete *key*
- update *key value*
- concat *key1 key2*
- fetch *key*

3. `m` - total number of client requests

4. `client_req *req_list` - list of all the client requests received in the input.

5. `pthread_mutex_t output` - lock accquired by the client thread who wishes to print on the terminal

6. In the *main thread*, we take the input, initialize the locks, create m client threads and wait for each of them to complete.

7. In the *client thread*, we create a socket and connect to the server using the `get_socket_fd` routine. Then the thread sleeps till the time at which the request has been made. Next, we acquire the client_mutex and try sending the command to the server using the `write` system call. If this *fails*, we `return` from the client thread. If the write call succeeds, we try to receive the server response to the request using the `read` system call. `read` is a blocking system call thus the client thread blocks until there is something to read through the *socket*. If the read call succeeds, client thread acquires the output mutex and prints the server response. `request_id : thread_id : server_response`.

8. In the `get_socket_fd` routine, we create a socket using the `socket` system call, which returns the file descriptor(fd) of the new socket. We have defined the port number of the server as `8001`. We set the `struct sockaddr_in server_obj`. Next we connect the socket to the server using the `connect` system call.

SERVER PROGRAM

Variables

1. `struct worker` has two data members,

```
int id; //worker id
pthread_t worker_thread_id;
```

2. `n` - command line argument - number of worker threads in the *thread pool* - these 'n' threads are the ones which are supposed to deal with the client requests. Thus, at max 'n' client requests will be handled by the server at any given instant.

3. `worker *worker_list` - list of n workers.

4.`pthread_cond_t service_client` - condition variable on which the worker threads wait when the client request queue is empty. Whenever the server accepts a new connection request from a client, the main thread signals the worker threads waiting on the service_client condition variable.

5.`pthread_mutex_t queue_lock` - a mutex acquired whenever we push or pop a client request from the client request queue.

6.`queue<int> client_q` - client request queue - whenever the server accepts a new connection request from a client, it pushes the client socket fd in the queue and whenever a worker thread starts, if the queue is not empty it pops a client request and process it.

7.`struct dict_entry` - structure for a single entry in the global dictionary - it has the following data members

```
int key;
char value[MAX];
// a mutex acquired whenever the entry is accessed/modified
pthread_mutex_t dict_entry_mutex;
int is_present;
```

```
// 1 if the entry was inserted in this session
// 0 if the entry was never inserted or deleted
```

8. `dict_entry dict[MAX]` - common dictionary

Routines

9. In the *main thread*, we initialize the locks, condition variables, create *n* worker threads. Next we create the server socket using the `create_server_socket` routine.

10. In `create_server_socket`, the server required `2` sockets, one to listen to the clients' connection requests (`wel_socket_fd`) and one to communicate with the connected clients(`client_socket_fd`). Since `listen` and `read` are blocking system calls, we need separate worker threads. We create the first socket using the `socket` system call, `wel_socket_fd = socket(AF_INET, SOCK_STREAM, 0)`. Next, we initialize the structures `struct sockaddr_in serv_addr_obj, client_addr_obj` using the `bzero` system call and set the port number as `8001` and other data members. Next we `bind` the new socket `wel_socket_fd` with the `serv_addr_obj`. Now the server start listening to the clients' connection requests using the `listen` system calls, `listen(wel_socket_fd, MAX_CLIENTS)`. For every new connection request, we use the `accept` system call which returns the `socket fd` of the connected client, `client_socket_fd = accept(wel_socket_fd, (struct sockaddr *)&client_addr_obj, &clilen)`. We push the socket fd in the `client_q` and signal the `service_client` CV, in case any worker thread is waiting on the same. (Since queue is global data structure accessed and modified by multiple threads we use `queue_lock`).

11. In the `worker_thread`, we acquire the `queue_lock` and check if the queue is empty, if yes, the thread waits on the `service_client` CV, else it pops a socket fd from the queue and calls `handle_connection(client_sock_fd)`.

12. In `handle_connection` routine, the server reads the command sent by the client using the `read` system call and calls `handle_command(command, client_socket_fd)` in case of a successful read.

13. In `handle_command` routine, we parse the command and store the command type (*insert/delete/concat/update/fetch*) and related arguments in a 2D character array `arguments`. Based on the command type the worker thread, reads/writes the contents of the dictionary and send corresponding message to the client. Here is the block for `update` command :-

```
if (num_args != 3)
{
    cout << "Invalid Command" << endl;
    return;
}

int key = atoi(arguments[1]);
char value[MAX];
strcpy(value, arguments[2]);
```

```
pthread_mutex_lock(&dict[key].dict_entry_mutex);
if (dict[key].is_present == 0)
{
    cout << "Updation successful" << endl;
    send_string_on_socket(client_socket_fd, "Updation successful");
}
else
{
    dict[key].key = key;
    strcpy(dict[key].value, value);
    cout << "No such key exists" << endl;
    send_string_on_socket(client_socket_fd, "No such key exists");
}
pthread_mutex_unlock(&dict[key].dict_entry_mutex);
```

Since dict is a global data structure read and written by all worker threads, we acquire the `dict_entry_mutex` before accessing the `dict_entry`.

14. `send_string_on socket` uses the `write` system call to send the message to the client

```
int bytes_sent = write(fd, s.c_str(), s.length());
if (bytes_sent < 0)
{
    cerr << "Failed to SEND DATA via socket.\n";
}

return bytes_sent;
```
