Multi-Threaded HTTP Server

Jake Armendariz

Updated my design doc with my analogy. Also recently changed how I am counting number of valid & invalid requests to after the log is written. Before it was total sent, not total written and I think this could cause errors.

Goal

Build upon my exisiting http server by adding multiple threads, full logging and a health check of total/failed requests.

Analogy - Mystery House

You want to enter the Santa Cruz Mystery house. You walk in the door, take a number, and wait your turn. Only N people are allowed to be in the Mystery house at one time. Once a spot opens up you may enter the house, but you must go alone, only one person at a time. Some guests take their time and walk slowly, othes rush through, feel free to walk with others once inside the house. At the end of the mystery house there is a resturant, but you must save your seat in the reservation list before sitting down.

P.S. remember to eat your meal quickly, because the resturant is still in the house, we must be respectful for those still in waiting room.

- You are a http request
- The waiting room is the my request queue
- The entrence is my get-request function which removes people from waiting room
- The entire house up to the reservation list is reading, processing and sending the http request/response.
- The reservation list saves the space for each file in writing to the logfile, allowing multiple logs to be written at once but to seperate locations within the memory file.
- The resturaunt is the log file, multiple requests being written at the same time. Once they finish their meal, a spot opens up in the house for someone else.

Design

I used //http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html for parts of my thread structure. I used their file http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/thread-pool-server.c for the structure of a queue of requests.

To keep my program thread safe I only have two functions deal with intaking requests. add-request() continues to be locked as it adds my request to end of queue. and get-request, which will lock and remove the front of queue. I am redesigning my code to use a requestObject queue instead of my httpObject queue to reduce the size of queue. With the entire httpObject as queue, each node contained the buffer, response, log header space, and it was slowing down the runtime to allocate all of this memory every time a new request came in.

All processing of httpObjects occurs inside of my handle*request*loop. Which will create a message object, read in request, check for errors, carry out request and send to log file. All concurrently.

Setup

enviorment object
I wanted an enviorment object to keep track of how many threads, logfile and the port number. Just organized the code a little. Initialized with getopt() using the command line arguments.

```
struct enviorment {
    int N;
    char logfile[28];
    int port;
    bool haslog;
};
```

request object
Keeps track of the number of requests, and is used for my linked list queue of incoming requests. This is the waiting room of my analogy

```
struct requestObject {
    int id;
    int clientfd;
    int logfd;
    struct requestObject *next;
};
```

Shared variables
Head of the queue of requests to be taken. First in first out.

```
requestObject *requests;
```

Tail, last request, used to add in requests

```
requestObject *last_request;
```

The size of request queue

```
int queue_size;
```

Shared mutex variable for shared locking

```
pthread_mutex_t request_mutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
```

Shared condition variable, keeps threads from busy waiting

```
pthread_cond_t  request_cond   = PTHREAD_COND_INITIALIZER;
```

The request que will initially be set to NULL, queue*size = 0. Accept() will trigger the add*request function which will set the clientfd of a new httpObject and add to queue of waiting messages.

thread-requests_loop(void* data)

Threads are waiting in threadloop(), I have a condition variable to keep them from spin waiting in the case of 0 requests. Once a signal is sent, if number of requests > 0. It will call get_request() which will dequeue and provide the message at the front of queue. Inside of this loop it can then handle entire http request and write to log (more on that later)

data keeps track of the thread ID

while in the locked area, I read and process the response header. Obtaining filename, content-length and checking for errors. I create the response header for the client and the header for logfile with sprintf(). I increase global variable log_cursor by the strlen(response header) + 10 for \n========\n.

Then with this formula (on a good GET/PUT request) the body in log file is

```
log_cursor += message->content_length*3 + 9*((int)(message->content_length/20) + 1) + 10;
```

```
thread_loop(void* data) {
        struct requestObject *req;
        struct httpObject *message
        while(true){
        if(queue_size > 1){
            req = get_request_object()
            if(req != NULL){
                message = malloc(sizeof(httpObject)
                read_request(req->cliendfd, message)l
                proess_request(message)
                send_response(message)
                write_to_logfile(message->log_index)
                free(message)
            }
        }else{
            //keeps thread from waiting idily
            conditonal variable waits
        }
    }
}
```

add-request(struct requestObject *request)

After accept recieves an incoming message, it will lock, and send a new httpObject to addrequest(). add-request() enters the critical region. mutex*lock and add the queue of requests to be handled (linked list connected by requestObject), increasing queue*size and then signal (via condition variable). Unlock mutex.

```
add_request(int clientfd){
    Mutex_lock()
    struct requestObject *request = malloc(sizeof(requestobj);
    if(queue_size == 0){
        request_list = request;
        last_request = request;
    }else {
        //adds to end of the list
        last_request->next = request;
        last_request = request;
    }
    queue_size ++;
    Mutex_Unlock()

    condition signal
}
```

get-request()

if the size of the queue > 0, it will return the message at the front of the stack, else it will return null.

I have a mutex lock to ensure that saftey of queue. I keep the mutex locked while checking the size, removing the front and for queue_size -= 1. This and add request are the only interactions with the queue and queue size to keep the variables safe.

This is all done while locked to keep program thread safe and protect the log file from overwriting itself.

```
get_request_object(){
    requestObject request;
    mutex_lock()
    if(queue_size > 0){
        request = request_list;
        request_list = request_list->next;
        if (requests_list == NULL) {
            last_request = NULL;
        }
        queue_size--;
    }else{
        request = NULL;
    }
    mutex_unlock()

    return messsage;
}
```

create-log-header(httpObject message)

Calculates the size of each log entry. It will construct a header, via sprintf(), then I use this formula to calculate the size of hex digits.

I chose to caculate the size of each req, then

```
mutex_lock()
if(head or fail or content-length == 0){
   log_cusor += 10
}else{
    log_cusor += 3 * content_length + 9 * (int)(1+content-length/20 +1) + 10;
}
mutex_unlock
```

- 3* content-length for each hex and a space.
- 9 * (int)(1+content-length/20 +1) for each 8 byte and the \n character

- 10 for the \n=======\n

write-to-log(httpObject message)

Log-write is handeled concurrently. Inside message->log*index its given the starting location. I wrote a function called* `log-amount(struct httpObject *message, int amount)` *that will write the hex bytes and increase log*index as it runs. The header was already created in GET, so this function gets to be pretty simple to just read the amount from file or server until its empty, and continually write this. In a PUT request it reads from the file created, in a get request it will reread what was written. In a health check it will print out the header, message-id and error_count.

```
log_write(message){
   pwrite(message->log_index, message->header);
   if(!message->valid){
        return;
    }
    message->log_index += strlen(message->header);
    file = open(message->filename)
    while(amount < 0)
        amount = read(file)
        log-amount(message, file)
    }
    pwrite(message->logfd, "\n========\n", 10, message->log_index);
}
```

Healthcheck

Very simple, I just keep a message->id of most messages for the total, and I increment a variable inside of a thread safe part of get-request() function. If there is no logfile I return a 404. I keep track of this throught the filename of the request.

httpObject

Inside of the httpObject, I carry all the information to complete a request. I added 6 parts to handle threads. The log_index keeps track of where its writing in the log file. Id is for debugging purposes and also the total number of requests. The header is just to make it easier when writing to log file, I created it once when finding the size needed, and save it then inside of httpObject.

I realize that this object is too large, part of the cause is I wrote my code for hw2 with not handling the method_str but its necessary for logfile, also the response str isn't necessary but, it was for my original implemnetation. May clean up this code to make the object smaller after I get my mutlithreading perfect.

```
struct httpObject {
    int method;
    char method_str[5];
    char filename[28];
    ssize-t content_length;
    bool valid;
    char buff[BUFFER_SIZE];
    char response[100];
    int status_code;

    int logfd;
    ssize_t log_index;
    char header[100];

};
```

socket/file functions - will add definitons of others later (in notebook)

`int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`

- pthread_t thread is the name of the thread to be started.
- const pthread-attr-t attr points to a pthread_attr-t structure created with pthread-attr-init(3) if NULL, pthread is created with default configuration.
- start_routine is the name of the function that will start the pthread
- arg is an agrument (often an identifying number for the thread) ex: &a
- if successful returns 0 --- if error returns error number

`pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;` Recursive mutex, for n times locked, it will need n unlocks. keeps the program from running these lines of code concurrently.

`int socket(int domain, int type, int protocol)` Creates a server socket. It will return a file descriptor for a new socket, -1 if it fails

Sources

getopt:
- //https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html - //https://www.geeksforgeeks.org/getopt-function-in-c-to-parse-command-line-arguments/
Threads - http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/thread-pool-server.c