Design Documentation for `httpserver.c (ASSIGNMENT 2)`

Jay Montoya | jaanmont | 1742317

Introduction

This design documentation is an extension of the httpserver.c design from assignment 1. Most syntax such as function call specifics and design on handling general requests is omitted since I already have a working simple HTTP server. This design document will include the details of adding the following features to my server:

1. Multithreading to support multiple client requests concurrently.
2. Logging of requests.
3. Server health-checking mechanism.

Multithreading:

Start-up:

In my design, the **main()** method of **httpserver**.c serves two functions: server initialization and assignment dispatch. Upon startup, the **main()** method is responsible for the following:

1. **Create** variables needed for initialization and dispatch.
2. Use the getopt() function to **parse arguments** according to the spec.
3. **Start** the pool of worker threads and the logger thread if logging is enabled.
4. Set up for **listening and dispatching** connections to be handled by workers.

It is important to firstly define the structs used in initialization and their contents:

```
struct assignment {
    int assignment_fd;
    struct assignment* next;
};
```

The **assignment** struct is a simple representation of an assignment to be used in a linked list of assignments. An assignment contains one variable: the client socket descriptor as **assignment_fd**.

```
struct log_request {
    enum command_t command;
    char* res_name;
    size_t content_length;
    uint16_t status_code;
    char* first_line;            // for use with logging errors
    char* health_check_content;  // for use with logging health checks
    struct log_request* next;
};
```

If logging is enabled, worker threads fill out a log request as they work on their assignments. After completing the assignment or encountering an error, the log request is filled with the necessary information and sent to a linked list of log requests using the **pend_log()** function *(see the section on logging)*.

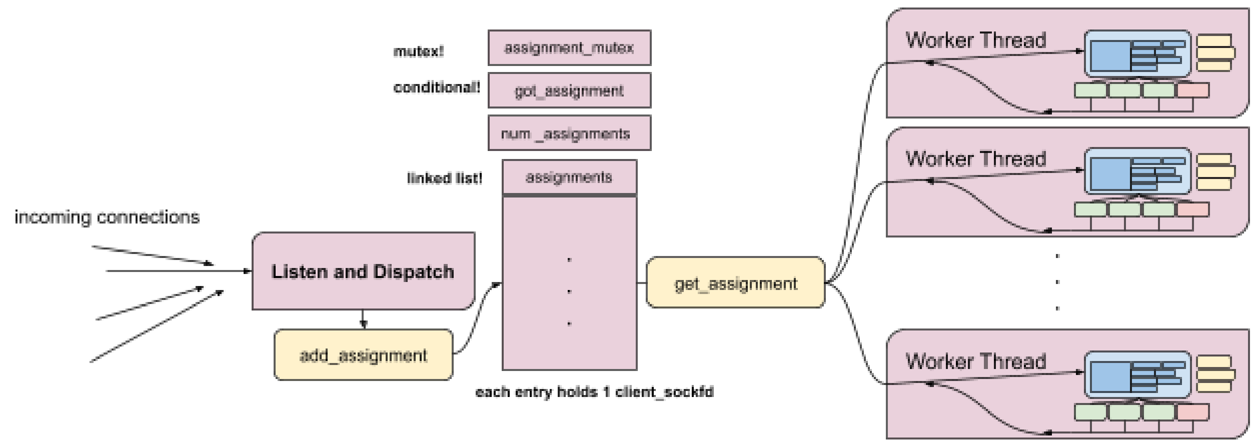
```
struct thread_args {
    int thread_id;                 // shared (read only)
    bool logging;                  // shared (read only)
    char* log_file;                // shared (read only)
    int* num_assignments;          // shared (read/write while holding mutex only)
    int* num_pending_logs;         // shared (read/write while holding mutex only)
    size_t* total_requests;        // shared (read/write while holding mutex only)
    size_t* total_errors;          // shared (read/write while holding mutex only)
    pthread_mutex_t* assignment_mutex;
    pthread_mutex_t* log_mutex;
    pthread_mutex_t* server_health_mutex;
    pthread_cond_t* got_log;
    pthread_cond_t* got_assignment;
};
```

This struct is passed to **every** worker thread. It contains the thread's id, the log file's name, the number of assignments, pending logs, total requests, errors, mutexes, and condition variables (these synchronization primitives are explained further in this section.

```
struct logger_args {
    int log_fd;
    char* log_file;                // shared (read only)
    pthread_mutex_t* log_mutex;
    pthread_cond_t* got_log;
    int* num_pending_logs;         // shared (r/w only while holding mutex)
};
```

This struct is passed to the logger thread. It contains important information for logging such as the log file descriptor, log file name, mutex and condition variable pertaining to logging, and the number of pending logs. It is important to note that the log file descriptor is opened/created/truncated in **main()** before it is handed to the logger thread.

Assignment Dispatch:



Topics:
dispatch thread (main)
assignments (linked list)
got_assignment (conditional)
assignment_mutex (mutex)
num_assignments (integer)
add_assignment (function)

Multithreading is achieved in this assignment through a dispatch thread (main) that will receive incoming connections and generate an assignment. An assignment is a struct that contains the client's socket descriptor. All assignments are added to a linked list.

This list is guarded by the mutex **assignment_mutex** which is a thread argument that will need to be locked in order to add or remove an assignment from the list. The number of total assignments in the list is also kept recorded in the size_t variable **num_assignments**.

After the dispatching thread creates an assignment, it signals this to the pool of worker threads by the conditional variable **got_assignment** (also passed as a thread argument). Threads block if there is no work to do and wait on this condition variable.

The dispatching thread makes use of the helper function **add_assignment()** which takes the client socket descriptor as **client_fd**, the mutex over the list as **p_mutex**, and the condition variable as **p_cond_var**.

```
void add_assignment( int client_fd,
                     pthread_mutex_t* p_mutex,
                     pthread_cond_t* p_cond_var,
                     int* num_assignments );
```

The function is then responsible for the following steps:

1. Create an assignment struct, which contains the integer socket descriptor for the client.
2. Lock the mutex **assignment_mutex** using **pthread_mutex_lock()**.
3. Add the assignment to the linked list of assignments (while incrementing **num_assignments**).
4. Unlock the mutex **assignment_mutex** using **pthread_mutex_unlock()**.
5. Signal the condition variable **got_assignment** using **pthread_cond_signal()**.

The only purpose of the dispatching thread (after initialization) is to relay accepted connections to the pool of worker threads.

Obtaining an assignment:

Topics:
worker threads (threads)
get_assignment (function)
perform_work(function)

On the other half of assignment multiplexing is a pool of worker threads that wait on the conditional **got_assignment** specified above. Threads are initialized before the receiving loop in main using **pthread_create()**, and each worker thread performs the following steps repeatedly forever:

- Lock the mutex **assignment_mutex**.
- If **num_assignments** is zero, wait on the condition variable using **pthread_condition_wait()**.

- After locking the mutex again: If **`num_assignments`** is greater than zero, call **`get_assignment()`** to obtain the next assignment. Unlock the mutex immediately after this call.
- Call **`perform_work()`**. This function will now fulfill the request as specified by asgn1 by using the client socket descriptor encapsulated in the assignment struct. perform_work() contains all code from asgn1 for unmarshalling arguments and executing requests. It is important to note that a **log_request** struct is returned by **perform_work()** and handed off to the logger thread with the **pend_log()** function if logging is enabled *(see section on logging)*.

Defined below are the function prototypes for **get_assignment()** and **perform_work()**:

struct assignment* get_assignment(int* num_assignments);

`get_assignment()` is a very simple function that returns the front object of the assignments linked list.

```
struct log_request* perform_work( struct assignment* an_assignment,
                                  int thread_id,
                                  char* log_file,
                                  size_t* total_requests,
                                  size_t* total_errors
                                  bool logging,
                                  pthread_mutex_t* server_health_mutex);
```

As explained above, **perform_work()** actually performs request interpretation, execution, and error handling. It contents are largely the same as found in **asgn1**, with the following modifications:

- **perform_work()** builds a log request as it goes, regardless of if logging is enabled or not. This log request is returned to the worker thread that called it after execution. The worker thread then decides, whether logging is enabled or not, to discard the log or append it to the list of waiting logs. A lot of this implementation is easy to implement as the fields in the log_request struct are already identified by my previous httpserver. For example, an assignment 1 line that reads:

  ```
  current_command = PUT;
  ```

  changes to:

  ```
  a_log_request->command = current_command = PUT;
  ```

- Health checking is executed in **perform_work()** *(see section on health checking)*. This design decision came about because of my logging thread being the bottleneck to accurate health readings. If the logger was responsible for updating health and was significantly behind in log requests, the server health variables would be stale data by a significant amount.

<u>IMPORTANT NOTE | CITATION:</u>

The design for handling multiple requests as shown in the above two sections is accomplished by emulating **thread-pool-server.c** from kent.edu. The URL of this code in its entirety can be found below:
<u>http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/thread-pool-server.c</u>

While the variable and function names in my design are very in-tandem with this design, there are a few fundamental iterations of this design that I've made.

- The design above made use of a recursive mutex, which I found to be very unportable, raising issues when multi-locking occurs. Instead, I changed this to a normal mutex.
- The loop for handling requests in **`thread-pool-server.c`** was not useful in my design, largely because the handling of a request is done while the thread still has the mutex locked. A significant change had to be made in this loop: A worker thread in my design obtains the mutex lock *ONLY* to get an assignment, this is crucial to increasing concurrency. The handling of a client when identified is unrelated to the assignment list, so the worker threads unlock the mutex immediately after obtaining the next assignment.
- With the above note, the **`get_assignment()`** function in my design does not deal with locks anymore as done in **thread-pool-server.c**
- **`add_assignment()`** in my design is very similar (if not the same) as **`add_request()`** in **`thread-pool-server.c`**. The reason being that it is nothing but simple linked list appending code which can be done in a few steps.

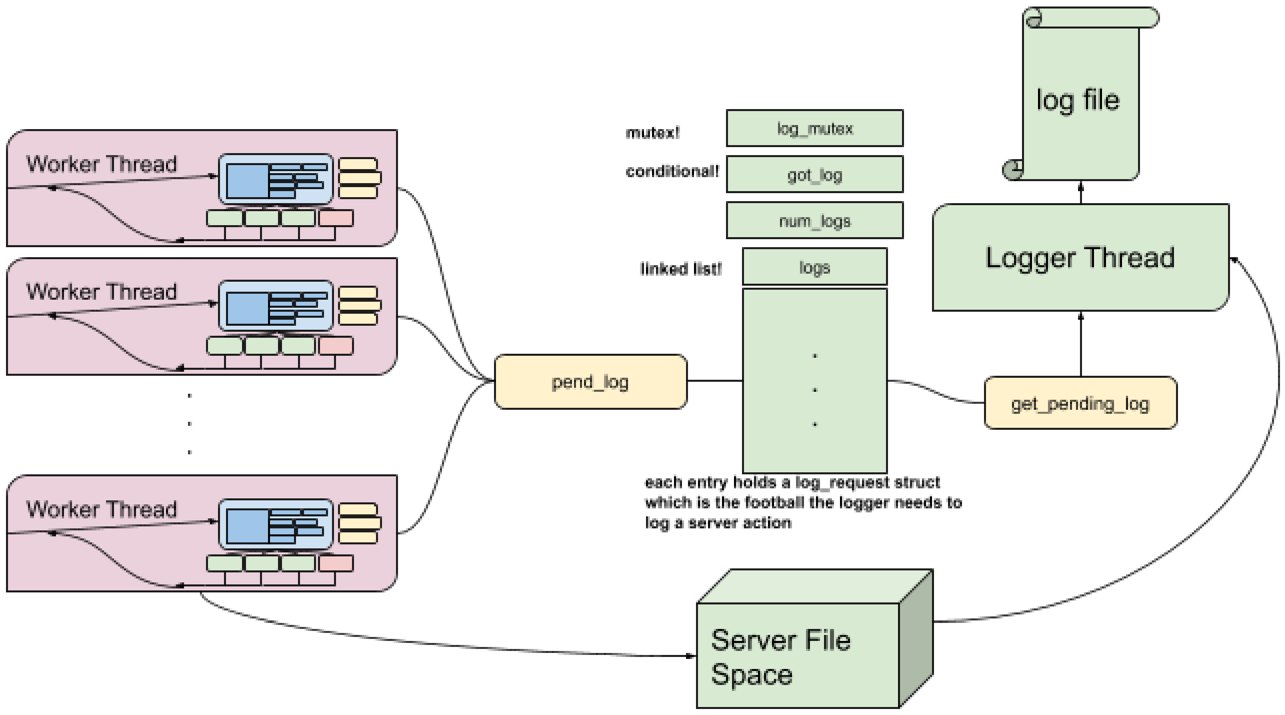
Logging:

Logging in my design is accomplished by using a separate "logger" thread to handle writing to the specified log file. This decision came about for many reasons, and the following points were considered:

- I want my clients to receive the same performance that they've seen in my single-threaded server. Primarily, it should not take a multi-threaded server longer to respond. Taking a **`pwrite()`** approach would burden the client by having them wait on a server log to be created *in addition* to their request being carried out.
- I do not want multiple threads writing to the log file: While a **`pwrite()`** approach may be possible with extreme caution, I want to streamline the logging process by only allowing one writer. Requests will be written to the log file *in the order that they were completed*.
- The general idea: Worker threads fulfill their requests, add a pending log to a list that the logger thread waits on. This list maintains the order in which threads complete their requests. The logger thread logs each request sequentially from the list and uses the resource in the working directory as it's log contents.

NOTE: This design decision has a drawback. Since the logger is essentially left on its own to catch up, the contents of each log may be different if a PUT has overwritten a resource that the log file intended on using but hasn't gotten to yet. This will result in undefined behavior if the server is extremely overwhelmed with many requests that overwrite one or more files. Since this edge case is not being tested, this server design will not support it. The log writer instead makes a best-effort to log the correct contents. To make up for this drawback, health checking and updating is done in the worker threads to maintain a correct server health status regardless of how overwhelmed the logger is.

Log Pending:



Topics:
pend_log (function)
get_pending_log (function)
log_mutex (mutex)
got_log (conditional)
num_pending_logs (int)

Just as assignments are multiplexed among the pool of worker threads, each worker thread builds a **log_request** struct that gets demultiplexed into a list of pending logs. Exactly as dispatch does, a worker thread (after locking the mutex **log_mutex**) will pend a log request using the function **peng_log()** whose signature is shown below. **pend_log()** adds the struct to the linked list, and notifies the waiting logger thread by signaling the condition variable **got_log**.

```
void pend_log( struct log_request* a_log_request,    // log request struct
                    pthread_mutex_t* p_mutex,         // log_mutex passed here
                    pthread_cond_t* p_cond_var,       // got_log passed here
                    int* num_pending_logs );
```

On the other side of the pending logs list is the logger thread, which waits on the signal from got_log if there is no work to do. The logger thread locks the log_mutex over the list, and then extracts the next log request using get_pending_log().

```
struct log_request* get_pending_log( int* num_pending_logs );
```

It is important to point out that before calling **get_pending_log()**, the log_mutex must be locked, as opposed to **pend_log()**, where the mutex is locked before the function call and released in the function call.

Logging and the Worker Threads:

Topics:
how does a thread build a log request?
edge cases in the worker threads

A worker thread utilizes **perform_work()** to build a log request as it goes, regardless of if logging is enabled or not. This log request is returned to the worker thread that called it after execution. The worker thread then decides, whether logging is enabled or not, to discard the log or append it to the list of waiting logs. Largely, this implementation was easy to implement as the fields in the log_request struct are already identified by my previous httpserver. All that needed to be done was export these fields to a field in the **log_request** struct in addition to the local variables of **perform_work()**. For example, an assignment 1 line that reads:

```
current_command = PUT;
```

changes to:

```
a_log_request->command = current_command = PUT;
```

There are some edge cases related to logging that needed to be covered by my design, they are presented as follows:

Case: When logging an error, the first line of the http request must be logged.
Solution: Added a char[] first_line in **perform_work()** to pass this value to the logger if an error occurs.

Case: A PUT request to the **log_file** should result in a 400 Bad Request.
Solution: Hardcoded check to compare the subject resource name and the log file (passed as a parameter to **perform_work()**.

Case: **perform_work()** does not deal with content_length of a GET or HEAD request (this is kept specifically to the functions **get_executor()/head_executor()**) but the **log_request** needs this field.
Solution: After performing a GET or HEAD request, the switch statement controlling execution in asgn1 needed to be equipped to "stat" the subject file in order to get the content_length. This had to be done, as these execution functions already return a value (the status code of the request).

Logging and the Logger Thread:

The logger thread, as mentioned before, is left on it's own to make a best effort to log the server's actions. The logger can receive 3 types of log request:

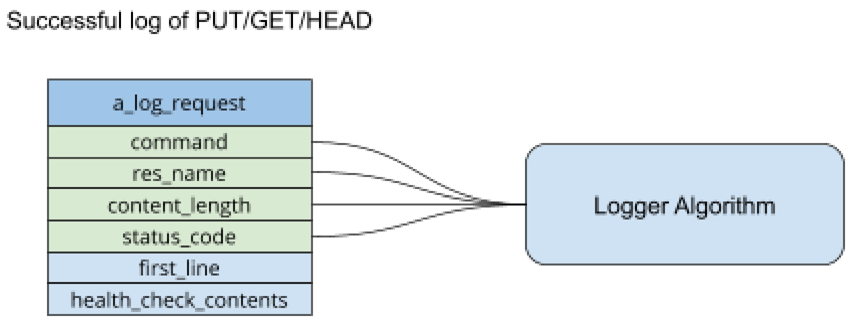
- A log request of success (PUT / GET / HEAD)
- An log request of an error (any 400, 403, 404, 500)
- A log request containing a health-check

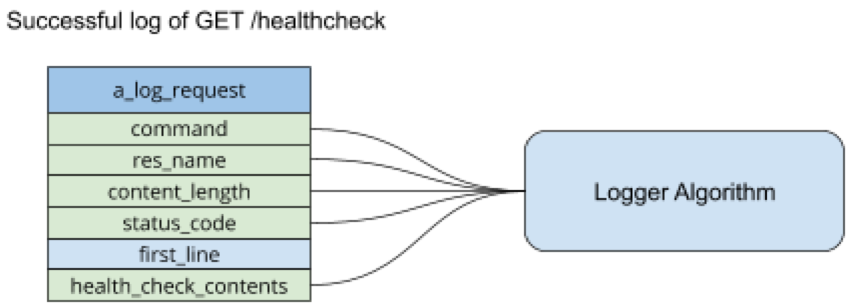
The following algorithm is repeated as long as the logger thread lives:

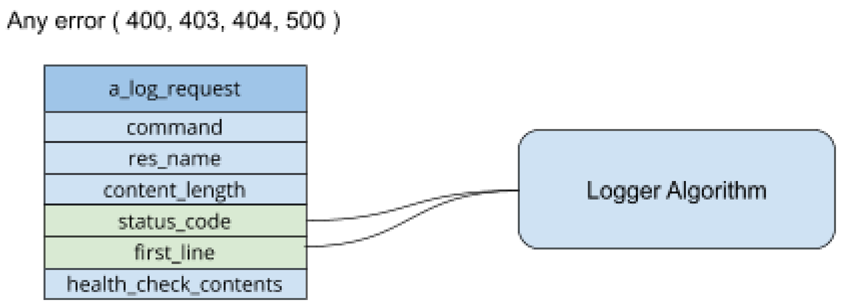
1. Get a log request using **get_pending_log()** or wait on the signal **got_log**.
2. If the resource name of the log request is "healthcheck" then set healthcheck to 1.
3. Deal with a failure first, if an error code is in the **status_code** field of the log request, use the **first_line** and **status_code** to generate a failure entry according to the spec. The log is generated using **strcat()** and written to the log file using **pwrite()**. Free the log request and continue to handle more logs.

4. Write the first line of the log (using the **command**, **res_name**, and **content_length** fields of the request). This line is built using a combination of **strcat()** and **sprintf()**, and written using **pwrite()**.
5. Open the file specified by the log request to begin reading data.
6. Determine how many lines will be needed for the content by using arithmetic on the **content_length**.
7. Read the subject file 20 bytes at a time, and use **sprintf()** to create the zero-padded markers on the left, as well as to create the hex dump of the data. Write the data line by line first into a buffer to format it, and then flush it to the log file using **pwrite()**.
8. If the healthcheck flag is enabled, the logger thread will instead read from the **health_check_content** field of the log request. Health check data is treated the same way as file data.

This algorithm requires careful attentivity to what the type of log request is, and how each step must be tailored to its specific request type. This concept of what the logger algorithm cares about depending on the type of log is demonstrated below:







Furthermore, it is the sole responsibility of the worker threads to assemble each log request in such a way that the logger algorithm will always be able to understand the type of log it needs to write.

Health Check

The health checking portion of this design is relatively brief, because the implementation of it is minimal. As stated previously, the reading and writing of the health check sharted variables **total_requests** and **total_error**s is performed by the worker threads.

Code for reading and writing of the health check variables is considered a critical section of code and as such is guarded by the **server_health_mutex**. Furthermore, a health-checking mechanism is implemented in my server by adding the following:

1. A critical region of the code in **worker_thread()**, where after the call to **perform_work()**, the **log_request** struct is inspected for an error or success code. The worker thread then locks the **server_health_mutex**, makes the modifications to these fields, and unlocks the mutex.
2. A critical region of code in **perform_work()**, where if the resource name requested by the client is “healthcheck” the function will lock the **server_health_mutex**, read from these fields, and unlock the mutex. A GET request to health check never makes it to the request execution functions, as the **perform_work()** function already has the resources (parameters) to handle this type of request.
3. A field in the log request struct contains the contents of a health check for the logger to interpret.

Once again, even for a health check request, the worker thread has the sole responsibility of constructing the log request in a way that it is obvious to the logger thread that the request contains health check data (in this case, by setting **a_log_request->res_name** to “healthcheck” as it would any other resource).

Final Notes:

<u>Why is this design deadlock free?</u>

This design is **deadlock free** because there is a **defined order** over how the resources will be accessed.
In a worker thread cycle, this order is as follows:

1. Assignments list (assignment_mutex)
2. Server health variables (server_health_mutex)
3. If enabled: Log request list (log_mutex)

Each thread must access these resources through the synchronization primitives in this order, additionally, each thread may only access **one resource at a time**, and never ask for another resource while already inside of a critical region. The resulting process-resource diagrams will therefore always be **directed acyclic graphs**, **attacking the “circular wait” condition** of a deadlock. Some possibilities are shown below: