Design Document: Multithreaded Web Server v1.0

Robert Hu

CruzID: ryhu

1.0 Introduction

1.1 Goals and objectives

The overall goal of this program is to modify the previous assignment's simple web server to allow for multiple threads, logging, and performing health checks.

1.2 Statement of scope

The main difference in this assignment compared to the previous assignment is the implementation of pthreads: a dispatcher to accept incoming connection requests and multiple worker threads, greater than 2, to handle the request and send it back. Other implemented features include logging to a specified file, only if the specific option is entered on the command line as well as performing health checks to check the failure rate of requests in the log.

1.2.1 Multithreading

The default number of threads in 4. The program accepts a command line option to change the value to another number (at least 2).

1.2.2 Logging

Logging only occurs if the logging option is specified on the command line. Logging keeps track of incoming requests and creates a log message for each request regardless of if they are successful or if they fail.

1.2.3 Health Check

Health checks are only valid for programs with the logging option. Otherwise, they return an error. They also are only valid with GET requests.

2.0 Data Design

2.1 Internal software data structure

A buffer is implemented to handle message passing within the main program to handle incoming messages.

2.2 Global data structures

In addition to the previous struct from the last assignment that held the httpObject, a new struct is formed in this assignment to handle the various new properties of the worker threads.

2.3 Database description

The struct httpObject has a new buffer in it to hold the log message that will be written to the log file with pwrite.

The new struct, worker, contains 8 variables. Since this struct is to be used during the creation of worker threads, it has an id var to hold its own id. It has a corresponding httpObject that is used when handling the request. It also has a client_sockd variable to remember which client sent the request. There are 3 pthreads: worker_id which is the actual worker thread, condition_var and lock for lock/unlock mechanics as well as waiting and signaling.

3.0 Architectural and component-level design

3.1 System Structure

The main design of the system for this assignment revolves around worker threads and a dispatcher. In my design, borrowing some help from Michael's section, uses the main function as the dispatcher.
The main function initializes an array of worker threads based on the provide or default thread number. Then it initializes the values of the variables for each worker thread in the array. In the while loop, the main accepts a connection, searches through the worker array for the first available worker, sets the appropriate variables and then signals them. The worker threads use a new function, **handle_task**, that handles the reading, processing, and returning the requests and log messages.

3.2 Description of Main function

3.2.1 Main function Responsibility

The main function checks the command line for the port number as well as checking for the -N and -l options. These arguments can occur in any order. The main also has added functionality to handle creating and initializing values for worker threads as well as assigning client socket ids to those threads and signaling them to begin work.

3.2.2 Main function Algorithm

```
INPUT: argc, argv
OUTPUT: none

1. Initialize threadNum, counters for requests, flags and file descriptors for
   logs and health check
2. Iterate through command line to check for -N and -l options and set the
   appropriate variables
3. (Create server socket from asgn1...)
4. Create array of worker structs and initialize offset and lock variables for
   dispatch
5. Iterate through array to initialize struct variables for each thread from 0
   to threadNum-1
    a. Check for pthread_create errors
6. while( TRUE )
    a. Accept incoming connection
    b. set targetThread to count % threadNum
    c. Iterate through threads
        i. Set first available target thread's client socket descriptor,
                    set availability flag to 0, and signal the appropriate
                    thread
                ii. if iterator var reaches end, wait on main condition
                    variable
                    1. set iterator variable to 0;
        d. increment count
```

Algorithm 1: main(...)

3.3 Description of Main helper function 1

3.3.1 Main helper function 1 Responsibility

The only new helper function in this assignment is the function handle_task. This function waits for a signal from the dispatcher (main) that a job is available and starts processing it. It handles the previous three helper functions from the previous assignment. It also handles writing to the log if the log flag is set

3.3.2 Main helper function 1 Algorithm

```
INPUT: thread
OUTPUT: none

1. Set stat struct and reference worker struct
2. While (TRUE)
    a. Check thread's client socket descriptor for valid number
        i. While descriptor is not valid, wait for a signal from
           dispatcher
3. Call previous assignment's three helper functions
4. Check if log flag set
    a. If status code is 200/201
        i. Call snprintf() to make log message
        ii. lock worker mutex, set log file offset, unlock worker
            mutex
        iii. If method is PUT, grab contents of the file that was
             "PUT"
        iv. Form the rest of the message in hex format, with 20
            characters per line and a 8 buffered counter to keep
            track of characters processed
    b. if status code is 400/403/404/500
        i. Call snprintf() to make FAIL: log message
        ii. lock worker mutex, set log file offset, unlock worker
            mutex
    c. Call pwrite() to offset with log message
5. Close client socket descriptor
6. set available variable to indicate availability to work
7. set client socket descriptor to less than 0
8. signal dispatcher
```

Algorithm 2: handle_task(...)

3.4 Changes to Asgn1 helper functions

3.3.1 Main changes

In read_http_response, some functionality is added to check for invalid methods as well as invalid HTTP versions. In addition to this, the function does a check to see if the filename for a request is "healthcheck". In this case, the function checks for errors and for appropriate conditions.. If there is none and the appropriate conditions are set, then it creates a file with the failure and total request count. The last change is regarding changing from strtok to strtok_r as the latter is thread safe.

In process_request, the only change is in status code failure checking.

In construct_http_response, the totalCount and failureCount variables are incremented based on the status codes of the requests.

3.3.2 Algorithm changes

```
INPUT: client socket, httpObject message

    1. (Original functionality.....)
    2. Check if valid method type
        a. If it is not valid, set code to 400
    3. Check if valid HTTP version
        a. If it is not valid, set code to 400
    4. Check if filename is healthcheck
        a. if method is not get, set code to 403
        b. if log flag is not set, set code to 400
        c. set healthcheck flag
        d. create/truncate healthcheck file
        e. write failure count ratio to file and close file
    5. (In header parsing)
        a. Change strtok() to strtok_r()
    6. (Rest of functionality....)
```

Algorithm 3: read_http_response(...)

```
INPUT: client socket, httpObject message

    1. Check if status code has been set to 400, 403, 404, 500
        a. Skip this function if this is the case
```

Algorithm 4: process_request(...)

```
INPUT: client socket, httpObject message

    1. Switch statement to compare and create status strings for each status
       code
        a. If status code is 400, 403, 404, 500 -> failureCount and
```

```
          totalCount increment
       b. If status code is 200, 201 -> totalCount increment
```

Algorithm 5: construct_http_response(...)