Design Document: Multithreaded HTTP Server

Andy Huang
CruzID: ahuang44

1 Goals

The goal of this assignment is to create a multi-threaded HTTP server in c/cpp. The main difference in this assignment over the last is that this will have multiple threads that can handle requests and will have logging. Logging will include both header information and the data dumped as hex. Will need to apply synchronization techniques to allow multiple threads to service multiple requests at once and write to the same log file. The server still needs to respond to GET and PUT requests. The server will write files with named by 27 character ASCII names. Server will persistently store files in a directory thus it is able to be run on any directory that may have files already.

2 Design

2.1-Initializing server with n threads from commandline specified ip:port and n threads
will use getopt() to handle command line. Program still requires that there is a specified IP/hostname
Assuming the same as previous assignment, the default port if not specified is port 80.

```
If no -n option, Default to n=4 and create n threads.
    Else create n threads
If there is no -l option, we don't have to do logging
    Else we have to do logging
```

2.1(a)-dispatcher and workers and semaphores:

Things we need to apply synchronization to:

1)dispatcher and workers need to sleep until awoken

2) synchronize logging between multiple threads so they can concurrently and continuously log.

tasks=init at 0. post() when dispatcher accepts a job and wait() when worker accepts a job
avail=init at nthreads. dispacher will wait() and worker will post() after work is done
mutex=binary semephore to allow access to workbuffer
offsetMutex= binary semaphore to allow access to offset for logging

-The dispatcher is the thread that gets started by the binary ./httpserver.
-It will create the n threads that we need either specified by command line or defaulted to 4.
-The dispatcher will accept() connections and try to pass the client’s fd to the workers through a shared buffer.
-the dispatcher sleeps if there are no available workers by a semaphore.
-The workers will all sleep until the dispatcher wakes them up through the tasks sempahore
-workers woken up will then take the mutex and get the client fd in the critical region
-worker will process the requests until the client closes the connection

-worker goes back to sleep and waits for dispatcher to give it another task(clientfd).

2.2-Processesing header

The second part is that we will need to process and parse the http header that gets sent to the server by http requests. Most likely will use sccanf() to parse the data that is in a string. The assignment specifies that names *must* be 27 ASCII characters long with only upper/lower case english alphabet, digits 0-9, dashes(-) and underscores(_). We are now required to allow resource names to be preceded by a "/".

If there is a "/", ignore it and make sure the rest of the name still holds the 27 character long with the allowed 64 characters.
Else proceed as before and verify the name is correct.

If this requirement is not met, we can simply reply with error (400 for bad request). For any errors while processing the header, we can reply with error (400 bad request). This will allow us to determine what kind of request it is and properly execute that request.
Using strstr() to look for "Content-Length:" so we can know how many bytes to read if its a PUT request
Using strstr() to look for \r\n\r\n to detect if we need to keep reading to get to the end of the header.

2.2(a)-PUT headers

There will be an empty line at the end of the header and this will signal that the header contents are processed and what comes after is the contents for a PUT request. Our http server now requires PUT headers to include a Content-Length: or else it is a bad request.

IF there is a "Content-Length: x" within the header, the program should only read "x" bytes after the header.
ELSE return and let the client know that this is a bad request because there was no content-length

2.2(b)-GET headers

Get headers should not have more than two lines because there should not be a "Content-Length" attached to it as the client does not know the length of the object they are requesting/"GETting"

2.3-functions for executing and responding to PUT/GET requests

The last part of the program is to execute PUT and GET requests when they are received and send a response.

2.3(a)-PUT

We will be using write() to create and/or write to files.

```
Open() the file.
If there is an error, check the error
        If the error is saying that there is no such file, create it with open()
Else no error, use truncate to make the file size 0 effectively getting rid of whatever is inside in case there is some weird case with not being able to overwrite the file.
```

```
If there is any error in writing to this file, we can reply with error(403 forbidden) along with the reason to signal that there was an error even though the request was valid.
ELSE if there are no issues,
        IF the file doesnt exist and we had to create it : (201: file created)
        ELSE we can reply with (200 OK) indicating there were no errors writing to a file that exists
```

2.3(b)-GET

We will be using read() to read from files in the same directory as the server

```
IF there is an error in reading the file,
    1) File not found reply with status 404
    2) we can reply with error(403 or 500) with the error message indicating what went wrong.
ELSE 1) we can reply with status 200 OK
        2) get the length of the file that we will be sending to properly create a response header with “Content-Length: x”.
        3) Fill our buffer with the contents of what we read() and send in reply after the status
```

2-4-Logging

Logging will be turned on by a global variable that gets set when the httpserver is run.
The file that needs to be logged to will also be a global int. Everyone can write to this same fd and should not change
We will need to use synchronization to keep track of the file offset that we will use to write to the logfile using pwrite().
Since threads must know their content length either from the header in PUT or filesize from GET, we can do the math to a global variable protected by some kind of synchronization technique like mutex locks.
Each thread will “reserve” their logging space after they process the header.
Each thread has to wait until no other thread is accessing the file offset.
When a thread is chosen to go to access it, it will take a copy of it and this is where this thread will start writing to the logfile.
It will then update the file offset by adding how much data it knows it will write to the logfile and leave the critical region allowing for the next thread to come around and know where to start writing and update the next write spot.