Design Document - Assignment 3

Derrick DeBose

1 Goal

The goal of this assignment is to create a HTTP protocol server in which we can use clients like curl in order to make get and put requests. The server will use caching in order to speed up memory access operations.

2 Assumptions

We are assuming that this code will be run on Ubuntu 18.04 VM. So, this code may not work if it were to run on a Mac or Windows system. We are going to assume that we are using an IPv4 address. Assume, no bad arguments will be given on the server side when starting up the server. Also assume that when starting up the server the host name is always given before the port number. We will also assume that multithreading will not be needed for this assignment.

3 Design

I will start the socket program by creating a server socket. I will then assign attributes of the address from the command line arguments like ip address and port number. We will be able to detect the -c and -l flags from the getopt function. I can then bind the address to server file descriptor. Then we will listen for a client to connect to the server.

Then we will start a while loop where we can accept the client's connection. Read the request from the client that we can parse the request on newline in order to figure out the command, the filename, the HTTP/1.1 header for the response header and the content length if the request was a put request.

Before we do anything with processing the request we want to first check to see if the status code is 500 by the command not being a get or a put request.

We will check to see if the filename used is invalid. If invalid then return a 400 error.

If there is no cache flag then we want to do our old get and put methods.

Otherwise we want to implement the cache on the get or put request.

If the request filename is in the cache and a get request: we want to get the file contents and content length from the cache and send the response and data over to the client.

If the request filename is in the cache and a put request: we want to delete the old put request in the cache and at the new request contents into the cache.

If the cache is full and a put request: then process the least recently used put request and remove the least recently used request from the cache and add the new put request into the cache.

If cache is not full and a put request: we want push that new request onto the cache.

Else we must have a get request that not in the cache so read from file and write to the socket.

Close the client socket, and continue to the top of the while loop in order to accept another connection from a client.

If we detect a ctrl+C and want to stop the server: process the cache put requests until the cache is empty. Lastly we need to close the server socket.

4 Pseudocode

We will use the Server algorithm as our main method where we will use the functions: get, put. We will need the global values: list cache, logFlag, logFile, and cacheFlag

```
if argc ==1 then
    error("Not Enough Arguments");
end
create socket file descriptor;
address.family = IPv4 address type;
if getopt(argc, argv, "c l:" then
    switch
     case c  cacheFlag = true
     case l  log = true
end
set logFile name;
set hostname or port number;
bind the server file descriptor to the address;
listen for a client;
while 1 do
    Accept client's connect;
    read client's request;
    split request on new line;
    scan lines for command, filename, and content length;
    if not get or put request then
        error(500);
    end
    if invalid filename then
        error(400);
    end
    if cacheFlag then
        if filename in cache then
            update cache and move cache item to the front;
            if get then
                process request based on content and content length in cache.front;
            end
            else
                pop front;
                repack the contents with new data;
                push new data to front;
            end
        end
        else if cache is full and a put request then
            put(list.back);
            pop back cache;
            pack contents of new request into struct;
            push front cache (new request);
        end
        else if cache is not full and a put request then
            pack contents of new request into struct;
            push front cache (new request);
        end
        else
            do a get request by reading from disk and not from cache
        end
    end
end
```

Algorithm 1: Server

```
Continuation from the previous algorithm
while 1 do
    if cacheFlag then
    end
    else
        if command is a get request then
            get content length;
            call get function;
        end
        else if command is a put request then
            call put function;
            print(content length);
        end
    end
    close client socket;
end
close server;
return 0;
```

Algorithm 2: Server continued

```
iterator iter;
for iter = cache.begin(); iter != cache.end(); ++iter do
    if strcmp(arguments.filename, iter.filename)==0 then
        arguments.filename = iter.filename;
        arguments.http = iter.http;
        arguments.sock_id = iter.sock_id;
        if strcmp(arguments.command, "GET") == 0 then
            arguments.content_length = iter.content_length;
            arguments.content = iter.content;
        end
        cache.push_front(*arguments);
        cache.erase(iter);
        return true;
    end
end
return false;
```

Algorithm 3: in_cache

```
if content_length >0 then
    char* content = new char[content_length];
    cache.content = content;
end
while content_length >0 && read_count >0 do
    unsigned char buf[50]; read_count = read(arguments.sock_id, buf, sizeof(buf))
    ; strcat(arguments.content, (char*)buf);
    content_length -= read_count;
end
if in_cache(arguments) then
    header [was in cache];
end
else
    header [was not in cache];
end
int fd2 = open(logFile, O_WRONLY — O_APPEND);
number = write(fd2, hexbuf, num);
uint64_t counter = 0;
uint64_t log_count = 0;
while content_length >0 && counter <= arguments.content_length do
    unsigned char buf[20];
    read_count = 20;
    for int i=0;i<20;++i do
        if counter==arguments.content_length then
            read_count = i;
        end
        if counter <= arguments.content_length then
            buf[i] = arguments.content[counter++];
        end
    end
    num = sprintf(hexbuf, "08lu ", log_count);
    number = write(fd2, hexbuf, num);
    for ssize_t i =0; i <read_count; ++i do
        num = sprintf(hexbuf, "02x ", buf[i]);
        number = write(fd2, hexbuf, num);
    end
    content_length -= read_count;
    log_count +=20;
end
num = sprintf(hexbuf, "========");
number = write(fd2, hexbuf, num);
close(fd2);
if status== 201 then
    201 Created;
end
else
    200 OK;
end
```


Algorithm 4: Pack Content

```
signal to detect ctrl+C;
while cache.size() not equal 0 do
    struct cache_arguments params = &cache.back();
    put(params);
    cache.pop_back();
end
close(server_fd); exit(0);
return 0;
```

Algorithm 5: FreeCache

```
open the file;
if 404 status then
    return 0;
end
else if 403 status then
    return 0;
end
while read_count >0 do
    read_count = read(file);
    if read_count >0 then
        content_length += read_count;
    end
end
return content_length;
```

Algorithm 6: get_filesize

```
open the file;
if in_cache() then
    write from content buffer to the file on the server;
end
else
    while content length >0 do
        read the file;
        sent count = send the file to client socket;
        content length -= sent count;
    end
end
```

Algorithm 7: Get

```
open/create the file;
if in_cache() then
    write from content buffer to the file on the server;
end
else
    while content length >0 do
        receive the data from the client socket;
        write count = write to the file on the server;
        content length -= write count;
    end
end
```

Algorithm 8: Put