Design Document - httpserver.cpp

Anthony Chian
CruzID: `achian`

CSE 130, Fall 2019

1 Goal

The goal of this program is to implement a server that can receive messages from a client and respond appropriately to either read from a file on the server (GET) or write to a file on the server from content that the client provides (PUT). This modified version of httpserver.cpp can also use multithreading, being able to handle multiple requests concurrently and also provides logging for each request, providing a history of the requests made by the client to the server. The logging translates the content to hexadecimal and also specifies the method and status of the request. Apart from that, this version also offers caching, being able to add/update page requests and read them from cache instead of disk

2 Assumptions

I'm assuming the user will run the server file with a hostname and port number. I'm assuming that the client will send content headers similar to the ones used with curl that I used for testing purposes. I'm assuming that the files sent can have any data, even null. I'm assuming that the user will either specify amount of threads with -N or not (default is 4), and will specify a file to log with -l or if not the program does not do logging. I'm assuming that the user will either include -c for caching or not if the user does not want caching

3 Design

The approach that I'm taking is to have a dispatcher thread listen for messages from the client in a while loop. The dispatcher thread is handles the requests that come in and signals a worker thread to wake up. The worker threads are all asleep and wait for a signal from the dispatcher. Each thread has a mutex lock to make sure there are no race conditions. Each worker thread is independent and once it finishes it's job it goes back to waiting for another request from the dispatcher. For every request the worker thread does there are multiple cases. For the case of an invalid file name, the server sends a Bad Request response. For the case of a non GET or PUT method the server sends an Internal Service Error. Then for a missing file name the server sends a Not Found response. For access to a private file the server sends a Forbidden response. When the server creates a new file successfully it sends a Created response, and when it overwrites or retrieves from an existing file it sends an OK response. The worker thread receives the socket from the client and retrieves data or uploads data. To avoid race conditions we also have a mutex for the socket. For the logging we use a mutex for the offset variable to allow each thread to write to the logging file concurrently. We also convert the data to hexadecimal and format it before finally writing to the calculated offset by each thread. If there is an error we log the error instead with the same method. For the case of caching, I am doing a FIFO linked list implementation of size 4. The cache starts empty, and can receive up to 4 pages. For a PUT request there are 4 cases. First, if the page does not exist in the cache, it is added. Second, if the cache is not full but the page exists in the cache, it is updated. Third, if the page is present in the cache and the cache is full, it is updated. Lastly, if the page is not in the cache but the cache is full, it is added and the oldest page in the cache is removed. For logging the logged request indicates if the page was present in the cache or if it was not.

4 Pseudocode

This is the pseudocode for the program.

Globals vars:

```
#define CAPACITY 4
int new_socket;
int offset = 0;
int numPages = 0;
bool logging = false;
bool caching = false;
const char *logfile;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t socketMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t offsetMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t threadCondition = PTHREAD_COND_INITIALIZER;
pthread_cond_t socketCondition = PTHREAD_COND_INITIALIZER;
```

Structs:

```
struct threadInfo
{
   int index;
   int socket;
};
struct Node
{
   char *file;
   char *content;
   struct node *next;
};

struct Node *head = NULL
struct Node *current = NULL
```

<u>Helper func:</u>

```
void printCache()
    struct Node *temp = head
    if (head == NULL)
        cache is empty
    else
        while (temp!=NULL)
            print temp->file, temp->content

void removeFirst()
    if (head!=NULL)
        struct Node *temp = head
        head = head->next;
        free(temp)
        numPages--

void addNode(filename, data)
    struct Node *new_node = malloc(sizeof(struct Node))
    new_node->file = malloc(sizeof(char) * len(filename))
    strcopy(new_node->file, filename)
    new_node->content = malloc(sizeof(char) * len(data))
    strcopy(new_node->content, data)
    new_node->next = NULL
    numPages++

    if (head == NULL)
        head = new_node
        return
    current = head
    while (current->next !=NULL)
        current = current->next

    current->next = new_node
```

```
bool updateCache(filename, data)
        if (head == NULL)
                addNode(filename,data)
                return false

        current = head
        while (current != NULL)
                if (filename == current->file)
                        current->content = data
                        return true
                current = current->next

        if (numPages == CAPACITY)
                removeFirst()

        addNode(filename, data)
        return false

char readFromCache(filename)
        if (head == NULL)
                return "none"
        current = head;
        while (current != NULL)
                if (filename == current->file)
                        return current->content
                current = current->next

        return "none"

bool isValid(file)
        if file length != 27
                return false
        if (loop for length of file)
                if ( !isalpha(file[i]) and !isdigit(file[i]) and file[i] != '-' and file[i] != '_' )
                        return false
        return true
```

```
**void cat**(fd, s)
    Declare buffer of size 4000
    Declare response of size 100
    Declare nbytes

    While loop (nbytes = read(...) > 0)
        If (write (....) != nbytes)
            err(...)
    end loop
    If (nbytes < 0)
        err(...)
**void processFAIL**(filename, logfile, method, response)
    declare info[100]
    int threadbuffer;
    sprintf(info, FAIL...method, filename, response)
    int fd = open(logfile)
    if (fd > 0)
        pthread_mutex_lock(&offsetMutex)
        threadbuffer1 = offset
        offset += strlen(info)
        pthread_mutex_lock(&offsetMutex)
        pwrite(fd, info, strlen(info), threadbuffer)
        close(fd)
```

```
void processGET(fd, file, socket, logfile)
    declare response[100]
    if file is not found
        write request 404 Not Found
        if (logging)
            processFAIL(...404)
    else
        found = false
        if  (caching)
            char buf = readFromCache(filename)
            if (buf != “none”)
                found = true
                write request 200 OK
        fd = open (file…)
        if (fd > 0)
            if (!caching || (caching && !found))
                cat (fd, file, socket)
                close(fd)
            if (logging)
                char info[100]
                int threadbuffer
                if (found)
                    write request 200 OK was in cache
                else
                    write request 200 OK was not in cache
                fd = open (logfile…)
                if (fd >0)
                    pthread_mutex_lock(&offsetMutex)
                    threadbuffer = offset
                    offset += strlen(info)
                    pthread_mutex_lock(&offsetMutex)
                    pwrite(fd, info, strlen(info), threadbuffer)
                    close(fd)
        else
            write request 403 Forbidden
            if (logging)
                processFAIL(...403)
```

```
void processPUT()
    declare data[4000]
    declare response[100]
    int fd
     ssize_t numBytes

    numBytes = receive (socket, data, size of data)

    bool updated;
    if (caching)
        updated = updateCache(filename, data)

    if file is not found
        write request 201 Created
    else
        write request 200 OK



    fd = open(...)
    if (fd > 0)
        write(fd, data, size of data)
        if (logging)
            char info[100]
            char hex[12000]
            int threadbuffer1, threadbuffer2, ncount, length
            for ( i to length of data)
                if ( i % 20 == 0)
                    length += sprintf(hex+length, "...", ncount, data[k]
                    ncount += 20
                else
                    length += sprintf(hex+length, "...", data[k]
            length += sprintf(hex+length, "\n========\n")
        if (updated)
            sprintf(info, "PUT…was in cache.", filename)
        else
            sprintf(info, "PUT…was not in cache.", filename)


        int fd = open(logfile)
        if (fd > 0)
            pthread_mutex_lock(&offsetMutex)
            threadbuffer1 = offset
            offset += strlen(info)
```

```
                threadbuffer2 = offset
                offset += strlen(hex)
                pthread_mutex_lock(&offsetMutex)
                pwrite(fd, info, strlen(info), threadbuffer)
                pwrite(fd, info, strlen(hex), threadbuffer2)
                close(fd)

    else
        write request 403 Forbidden
        if (logging)
                processFAIL(...403)
```

<u>Dispatcher thread function</u>

```
dispatcher(NTHREADS, hostname, PORT, threadInfo threads[])
    declare struct sockaddr address
    declare serverfd, socket, fd….

    address.fam = …
    address.port = port
    address.hostname = hostname


    if (bind(…))
        error()
    if (listen(...))
        error()

    while(1)
        if (socket = accept(...))
            error()
        pthread_mutex_lock(&mutex)
        pthread_cond_signal(&threadCondition)
        pthread_mutex_unlock(&mutex)

        pthread_mutex_lock(&socketMutex)
         while (new_socket != -1)
            pthread_cond_wait(&socketCondition, &socketMutex)
        pthread_mutex_unlock(&socketMutex)
```

Worker Thread:

```
void *workerThread(void *threadArgs)
    struct threadInfo *data
    data = (struct threadInfo *) threadArgs
    while (1)
        pthread_mutex_lock(&mutex)
        pthread_cond_wait(&threadCondition, &mutex)
        pthread_mutex_unlock(&mutex)

        pthread_mutex_lock(&socketMutex)
        data->socket = new_socket
        pthread_cond_signal(&threadCondition)
        pthread_mutex_unlock(&socketMutex)

        char buffer[4000]
        filename = substring(buffer)
        method = substring(buffer)

        recv(data->socket, buffer, sizeof buffer, 0)
        print(buffer)


        if(isValid(filename)
            write request 400 Bad Request
            if (logging)
                processFAIL(...400)

        else if (method != “GET” and method != “PUT”)
            write request 500 Internal Service Error
            if (logging)
                processFAIL(...500)
        else
            if (method == “GET)
                processGET(fd, file, socket)
            else
                processPUT(fd, file, socket)
        close(socket)
```

Main func:

```
main(argc, argv[])
    hostname = argv[1]
    port = argv[2]

    for (i to length argc)
        if (argv[i] == "-1")
            logging = true
            logfile = argv[i+1]
    for (i to length argc)
        if (argv[i] == "-c")
            caching = true


    int NTHREADS = 4
    for (i to length argc)
        if (argv[i] == "-1")
            NTHREADS = argv[i+1]

    pthread_t threads[NTHREADS]

    struct threadArgs thr[NTHREADS]

    for (i to NTHREADS)
        thr[i].index = i
        thr[i].socket = 0
        pthread_create(&threads[i], NULL, workerThread, thr[i])
    dispathcer(NTHREADS, hostname, PORT, thr)

    return 0
```