Assignment 3 Design

I. Introduction

A. Objective

The goal of this assignment is to implement an HTTP server that responds to GET and PUT commands to read and write files, respectively. This server will have two new features: backups and recovery.

B. Statement of Scope

The HTTP server will listen for GET or PUT requests, followed by a 10 character ASCII file name and, optionally, the content-length on a user-specified port. A GET request to the server followed by a filename will return the contents of a file that is stored persistently in the same directory or return an error if the file is not found. A PUT request to the server, followed by a file name and data will send the data to be put into a new file denoted by the file name. Each response header to a GET or PUT request will be a valid HTTP response that specifies the HTTP version, the status code, and content length.

When the HTTP server receives a special request to GET a file named "b", it will create a new folder called "./backup-[timestamp]", where [timestamp] is the time when the backup was created. A copy of all files in the HTTP server folder will be created and stored in this backup folder.

When the HTTP server receives a special request to GET a file named "r", it will restore the most recent backup by copying all files from the newest backup folder to the HTTP server folder. If a timestamp is specified after the filename r (r/[timestamp]), the server will restore the backup with the provided timestamp. If no such backup exists, then an error 404 is returned.

When the HTTP server receives a special request to GET a file named "l" (lower case L), it will return a list of timestamps of all available backups, separated by a newline.

II. Restrictions

The program will not utilize standard libraries for HTTP or any of the C library FILE* functions except for when printing to the screen. The system calls: read and write are used for file I/O. The other system calls: socket, bind, listen, accept, connect, send, and recv are used for socket programming. The port numbers that the user can specify are restricted to port numbers above 1024 unless the server is run with sudo. Each thread is also only allowed to be allocated 16 KB of buffer space.

III. Program Summary

1. Global declarations:

- **a.** Struct **HttpObject**
  - **i.** Stores a parsed **request** from **commQ**
  - **ii.** Used by almost all functions in ***workerThread()**

b. Struct **Flags**
    i. Defines specific conditions for a **HttpObject**
    ii. Used by almost same functions as **HttpObject**

2. Start in **main():** The main initialization/setup before accepting HTTP requests
    a. Obtain the host address with **argv[optind]**. The host address should be the "first" command line argument after getopt() is called.
    b. Determine which port number to listen from using the function **getPort()**, which takes in argc and argv[]. If **optind++ == argc-1**, then the port number is not specified and should be set to the default value, 80. Else, port number is specified in **argv[optind]**. The user-specified port must be above 1024 and the program will exit if an invalid port is provided.
    c. A **sockaddr_in struct** named **server_addr** is declared. **inet_aton()** is used to convert the first command line argument, which specifies a hostname or an IP address, to a network address for the socket. The port number is then assigned to the socket as well.
    d. The server socket is created with a file descriptor, **server_socket**, that refers to it.
    e. Socket options are set using **setsockopt()** with the option **SO_REUSEADDR** to allow the reuse of addresses and ports.
    f. **bind()** is used to assign **server_addr** to the server socket.
    g. **listen()** is used to listen for incoming connections on the server socket.
    h. A **sockaddr struct** named **client_addr** is declared.
    i. An infinite while loop will be implemented to receive requests from the client. This loop will continuously accept client requests and process each request with a call to **executeFunctions()**.

3. In **executeFunctions()**, the following will occur:
    a. HTTP header from the client socket will be received and stored in a buffer.
    b. **recv()** once from the client to get/parse the header
    c. Store the first line (header) into the request object **request**
    d. Verify that the HTTP version and filename requirements are correct
    e. If GET request then check to see if filename is b, r , or l
        i. Set the corresponding flag to true if b, r or l
    f. Continue parsing through the first recv for content length or 2 new lines
        i. Set a pointer to end of 2 lines to signify start of body for put request
    g. If the request is a **get_request():**
        i. Check to see if filename is *b*, *r*, or *l*
            1. If b copy folder contents into a new folder with the format backup-*timestamp*

2. If r without timestamp find the most recent backup and restore it; otherwise copy the backup containing the timestamp and restore it
3. If l go through the directory and list all the backups to the client
4. Send 200 ok response and return to **executeFunctions()**

ii. The HTTP response is constructed and sent to the client, then the file will be continuously read and sent to the client until the end of the file is reached.

iii. The file is then closed before returning to **executeFunctions()**.

4. If the response is a **put_request():**

a. The file specified by the filename in the header is checked to see whether it already exists or not using **access()**. The file path provided to access() is obtained through the function **pathName()**.

b. The flag **exists** in struct **flags** will be updated accordingly.

c. If the put request has no error, the file path is opened; if the open fails, return a 500 status code with the response through **syscallError()**. This is the appropriate status code since, unlike GET, if the named file does not exist, it should be created. This is because we use the **O_CREAT** flag in our **open()** syscall.

i. We don't want to create a file if there is an error with the put request but we have to **recv()** to clear the buffer for the next request regardless of valid or invalid

d. Then check whether the content length of the file was specified in the header (whether or not it is equal to 0).

i. If the content length is specified, the server stops receiving data after the specified number of bytes, constructs the HTTP response, sends the response to the client, closes the file, and then returns to **executeFunctions()**. If the content length isn't specified, the server receives data until the client closes the connection or the end of the file is reached.

IV. Pseudocode

A. struct httpObject:

```
char type[]
char filename[]
char httpversion[]
char body[]
char* collector
int status code
content length
```

END

B. struct flags:

bool exists

bool isFirstParse

bool nameIsValid

bool fileB

bool fileR

bool fileL

END

C. struct LinkedList:

int data

struct LinkedList* next

D. void append(head, newData):

Create new node

Create new pointer and point at head of list

Put newData into new node

New node's next pointer = NULL

If list is empty, new node is also head of list

Traverse list until last node is found

New node is now the new last node of the list

END

E. void sendList(comm_fd, LinkedList node):

WHILE node is not NULL

Format node data into string and add '\n' at the end of it

Send node data to comm_fd

Node = node→ next

END

F. void executeFunctions(comm_fd, httpObjects request, buf, flag):

Recv from client into buf

Sscanf from buf (type, filename, httpversion, flag) delimited by space into request

Check if header is valid or not

While loop for parsing through header

Parse for content-length

Store content length if found

If 2 new lines found set body pointer

Call Get or Put request depending on type

END

G. valid_name(filename, flag):

Remove '\' from front of file name

IF filename is not 10 char long

IF filename == 'b'

Exception, good_name = true

Return true

ELSE IF filename == 'r' or 'r/alldigits'

Exception, good_name = true

Return true

ELSE IF filename == 'l'

Exception, good_name = true

Return true

ELSE

good_name = false

Return false

IF chars are not all ascii

Return false

Return true

END

H. void parse_request(comm_fd, httpObject request, char buf, flag):

IF is first time calling parse

Check http version

Check if name is valid

(ELSE parsing for content-length)

IF content length string exists

Store into request.content length

ELSE (no content length string in header)

break and leave content length empty

END

I. void get_request(com_fd, httpObject request, buf, flag):

Memset buf since all objects are parsed into request already

IF get “b”

Create new folder “backup-[timestamp]”

Open cwd

WHILE(readdir != NULL)

Check whether file is a directory

If file is not a folder, go to next file

Check if filename is valid

copyFiles()

ELSE IF get “r”

IF timestamp specified

Append specified timestamp after “backup-”

WHILE(readdir != NULL)

Check whether file is a directory

If file is not a folder, go to next file

Check if filename is valid

copyFiles()

ELSE recover most recent backup

Int newtime = 0

WHILE (readdir != NULL)

Check whether file is a directory

IF file is not a directory, go to next file

IF directory name starts with “backup-”

Extract timestamp from name and convert to int

IF timestamp > newtime

Newtime = timestamp

WHILE(readdir != NULL)
Check whether file is a directory
If file is not a folder, go to next file
Check if filename is valid
copyFiles()

ELSE IF get “l”
Create new list
WHILE(readdir != NULL)
IF file is not a backup, continue
IF file is a directory named “backup-...”
Extract timestamp from name
Contentlength += strlen(timestamp) + 1
Append timestamp to list
request->content_length = contentlength
construct_response()
sendList()
clearList()
RETURN

IF(fileR)
request→ status_code = 200
construct_response()
RETURN

IF(fileB)
Request → status_code = 201
construct_response()
RETURN

Open file with file path for reading
IF error
Set status code to error number
Construct response
Send constructed response

ELSE (get request succeeds)
Set status code to 200
Get content length of file being read and store into request

IF rflag

Create the file paths for folders copy2 and copy3

IF two of the files from the folders copy1, copy2, or copy3 were successfully opened and compareFiles() returns true, set sendFile to open one of the identical files

ELSE

Set status_code to 500

ELSE

Set sendFile to the filename stored in the request struct

Construct response

Send constructed response

Read and send sendFile

Close sendFile

END

J. void put_request(com_fd, httpObject request, buf, flag):

Check whether file name already exists and set exists flag accordingly

If no error open file from filename for writing

Check to see if open is successful

If buf is not empty write first from buf to file

IF content length exists:

While content length is not 0:

Write from buf to file

Content length-bytes received

IF content length is 0 (meaning all of data was received)

IF file already existed

Set status code to 200

ELSE

Set status code to 201

ELSE(data was not all received)

Throw error 500

ELSE (content length does not exist)

While client does not close connection(recv != 0)

Read from buf to file until EOF

Construct response

Close file

END

K. void construct_response(comm_fd, httpObject request):

Char length_string[20]

Convert request→content_length from a long int to to a string and store in length_string

Char response[50]

Copy request→httpversion to response

Concatenate " " to response

Concatenate getCode(request→status_code) to response

Concatenate "\r\n" to response

Concatenate "Content-Length: " to response

Concatenate length_string to response

Concatenate "\r\n\r\n" to response

send(comm_fd, response, strlen(response), 0)

END

V. Testing

A. Classes of Tests

White-box testing will be utilized while testing the functionality of the program. Although the use of C library FILE* functions are restricted for the functionality of the program, they will be used during testing as a convenient way to debug and keep track of values such as loop iterations and filenames. These print statements will be removed before submission to ensure clean and readable code. The program will be incrementally developed, so these tests are performed at every stage. These tests are described as follows:

- **a.** Test if **get_request()** can interpret b, r (with and without timestamp), and l
- **b.** Test backup
  - **a)** Test to make sure that path names are correct
  - **b)** Test to see if files are copied from origin folder to destination folder
- **c.** Test recovery
  - **a)** Test with timestamp
  - **b)** Test without timestamp
    - **(1)** Make sure to get the most recent backup
  - **c)** Test to make sure that path names are correct
  - **d)** Test to see if files are copied from origin folder to destination folder
- **d.** Test list
  - **a)** Test if all directories can be listed

Black-box testing will also be utilized through manual testing. The code will be tested using a variety of different combinations of inputs. Many curl requests were compiled together in a shell script. The script runs the clients and sends requests back to back in the background (the curls have "&" at the end of the command). The tests include the following:

- **a.** Test **GET** and **PUT** requests
  - **i.** A valid get request
  - **ii.** A get request with invalid files
    1. Nonexistent file
    2. Invalid name
       - **a.** File name with a non alphanumeric character
       - **b.** File name less than 10 ASCII
       - **c.** File name more than 10 ASCII
  - **iii.** Files with no read permission
  - **iv.**

b. Test b, r, l

- **i.** Test if /b creates a folder and puts files inside
- **ii.** Test /r requests without timestamp
- **iii.** Test /r requests with timestamp
- **iv.** Nonexistent folder
- **v.** Invalid name
    1. folder name with a non alphanumeric character
    2. folder name less than 10 ASCII
    3. folder name more than 10 ASCII
- **vi.** Test /l to see if it lists all backups

B. Expected Output

a. When httpserver receives a valid **GET backup** ("/b") request, a new folder named "backup-[timestamp]", with the appropriate timestamp, is created. Then, all files with a valid name in the HTTP server will be copied into the new folder and a 201 is returned.

b. When httpserver receives a valid **GET recovery** request, one of the following cases will happen:

- **a)** The request specifies "/r", so the HTTP server restores the most recent backup by copying all files from the newest backup folder into httpserver's folder and returns a 200.
- **b)** The request specifies a timestamp (i.e. "/r/[timestamp]"), so the HTTP server restores the files of the specified backup folder. If no such folder exists, then an error 404 is returned.

c. When httpserver receives a valid **GET list** ("/l") request, the HTTP server will send a 200 before returning a list of all the available timestamps, each separated by a newline ('\n').

VI. Assignment Questions

A. How would this backup/recovery functionality be useful in real-world scenarios?

Backup and recovery is useful in almost every scenario since unexpected things will always happen. The ability to backup and restore versions substantially reduces the negative effects of these unexpected cases.

The usefulness of this backup/recovery functionality is really evident in real-world situations. Github is one example. Pushing a "snapshot" of your project is like creating a backup: it allows the potential of accessing your code from a previous point in time. Recovering a commit is like the recovery function: you're able to restore a previous, and hopefully working, version of it.

VII. Citations

A. Used append() code as reference:
https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/?ref=lbp

B.