Asgn3 Design Document
Prepared by: Miranda Eng & Shirley Phuong
CSE130: Principles of Computer Systems Design
December 11, 2020

Program Description:
This program implements a backup and recovery enabling, single-threaded HTTP server that responds to GET and PUT commands to read and write files named by 10-characters in ASCII. In response to a GET request from a client, httpserver will detect a file name, read the file contents, write the associated HTTP response header, and, if appropriate, the associated content to stdout on the client side. In response to a PUT request from a client, httpserver will also take data from the request then create or overwrite the client specified file to put into the server directory and write the associated HTTP response header on the client side. If a GET with a file name of "b" is detected, the server will backup all files that are 10 alphanumeric characters long to a backup folder. If a GET with a file name of "r" is detected, the server will recover the most recent backup folder and overwrite the files in the current directory. If a file name of "r/[timestamp]" is given, the server will recover that specific backup folder. If a GET with a file name of "l" is provided, then the server will list all of the timestamps from the backup folders and send it to the client.

Design:
Our program does not require any data structures, however we utilize non-trivial formulaic-like structures in our code. We will mention these formulaic-structures. But first, we layout our basic program logic:

Program Logic
1. Get command arguments
    a. Check number of arguments
2. Setup TCP connection
3. Get HTTP request
4. Parse HTTP request header
    a. Check syntactic requirements (e.g. 10 character file name)
5. Handle GET request
6. Handle PUT request

Functions: `getaddr()`, `getStatus()`, `sendheader()`

Additionals:
- Handle basic errors to reduce noise and produce more elegant warnings and error messages
- Locate conditions requiring associated HTTP response and handle accordingly
- Add error state variable to prevent duplicate errors from producing excess in pipeline

Program Logic:
1. Get Command Arguments
Simple if statements. If 2 arguments, set default port number (i.e. 80). If 3 arguments, set specified port number. If not 2 or 3 arguments, ask the user to re-enter with proper usage syntax.

2. Setup TCP Connection
We follow the same familiar TCP client-side logic of `socket()`, `bind()`, initializing `servaddr`, `bind()`, `listen()`, `accept()`, `close()`, then looping back to `accept()`. Check for return values from these functions and handle them accordingly.

3. Get HTTP Request

To get the header of a HTTP request file, we use `recv()` to get one char at a time from the `commfd` socket. We receive in units of one character to prevent from reading past the message length. As we loop the `recv()`, we concatenate each character to a bigger string holding all of these characters. Once this larger string contains the carriage return `\r\n\r\n` (we detect using `strstr()`), that means we have reached the end of the header and must break to stop `recv()`.

4. Parse HTTP Request Header

In order to obtain parts of the header such as the request type and the file name, we used the `strtok()` function. This grabs the first two tokens in the header.

```
request = strtok(header2, " ");
path = strtok(NULL, " ");
```

From there, we can parse the path in order to obtain the timestamp for a specific recovery if there is one.

```
filename = strtok(path, "/");
timestamp = strtok(NULL, “/”);
```

To catch errors made by the client side, we use a multitude of if statements. In order to check if the file name is alphanumeric, a for loop is used to go through the file name and `isalnum()`, defined in the ctype.h header file, checks if the char is either an alphabet or a digit. To make sure the request type is HTTP/1.1, `strstr()` is used. The first parameter takes in the header and the second parameter takes in the string “HTTP/1.1” and looks for it in the header.

```
for (loop through file name) {
      if (!isalnum()) {
            // 400 Bad Request
      }
}

char* ptrhttp = strstr(header, "HTTP/1.1");
```

Other errors are also checked in the beginning such as the correct request type as well as the length of the file name provided (if they are not “b”, “r”, or “l”). These are just checked with the `strlen()` and `strcmp()` functions. If we get any of these errors, the response is created and sent with our sendheader() function. We also check if the client sends a specific recovery request but doesn’t provide a timestamp. This is done by checking for “r/” using strstr() and then checking if a timestamp is given.

```
char *rpointer = strstr(path, "r/");
char *filename = strtok(path, "/");
char *timestamp = strtok(NULL, "/");
if (rpointer != NULL && timestamp == NULL) { send 400 }
```

In addition to this, we include an integer called “error” to keep track of any errors we run into. This is to make sure multiple responses are not sent if there are multiple errors in the request. Two defined macros are created:

```
#define ERROR 1
#define NO_ERROR_YET 0
```

Once the server runs into an error we set the integer to 1 so that it cannot go into another if statement if the request contains more than one error.

5. Handle GET Request

Our code structure for handling GET requests is not very noteworthy, however it is a very important part of the program. We use if statements to check whether the request is to backup, recover, or list all of the timestamps. If it is none of those, we perform a regular GET request.
If the request has a file name of "r" and the timestamp is provided, we recover using the specific timestamp. We concatenate the timestamp to a backup string so that we can open that specific directory. At this point, we have to check if that directory exists or if it has reading access. We send a 404 and 403 response to these cases accordingly.

```
if (ENOENT == errno) { send 404 }
if (EACCES == errno) { send 403 }
```

If the backup directory exists and we have reading permission, we go through the specified directory and obtain the names of the files in the recovery folder. We open the directory using the dirent header file.

```
struct dirent *dp;
DIR *pdir = opendir(directory);
```

After getting the name using `dp->d_name` we can obtain the file's path using sprintf(). We also need to check if the recovery file has reading permission and the main directory's file has writing permission. If all of this error checking passes, we can open the files. We use a while() loop to read the recovery file and write to the current directory's file. We send a 200 response once this is finished.

If the client sends a recovery request for the most recent backup, we perform similarly to the specific recovery. We open the main directory and go through all of the backup directories. If the directory name has "backup-" in it, we obtain the timestamp by moving our pointer and then turning the timestamp into an integer. Once we get the integer, we can compare all of the timestamps and use the max as the most recent backup. Since we use a pointer to point to the name of the directories, we can assume that if the pointer is still NULL by the end, there are no backup folders in existence. In that case, we can send a 404 response.

```
while (go through directory)
      if (strstr(directory, "backup-"))
            set pointer to directory name and move to obtain timestamp
            convert timestamp to int
            if (timestamp > max) { set max to timestamp }
```

From there, we can just open the directory with the max timestamp number similar to what we did before. If the directory is not accessible, we send a 403 response. Otherwise, the code to recover is the same as the previous recovery code. We check for reading and writing permission and then use a while loop to read the recovery file and write to the current directory's file. At the end, we will send a 200 response.

For a backup request, we use mkdir() to create the directory and time() to obtain the time in seconds since January 1, 1970. To create the name of the directory, we concatenate the string "backup-" and the time() result. We can then use this name in mkdir() to create the directory. From there, we use dirent to go through the main directory and copy all eligible files into the backup folder. In order to do this, we check if the file names are alphanumeric and if they are 10 characters. We skip all files that do not fit these parameters as well as files that do not have reading permissions or end up being directories. Once we find an eligible file, we open the file and then create one in the backup folder. We use a while() loop to read the file in the current directory and write to the newly created file. Once we are done, we can send a 200 response.

```
while (go through current directory)
      if (file name is not alphanumeric, is directory, length is not 10,
```

```
                no reading access)
            continue;
        sprintf(d, “./%s/%s”, backup, file name)
        infd = open(file name, O_RDONLY)
        getfd = open(d, permissions)

        while (read(infd, buf, sizeof(char)))
            write(getfd, buf, sizeof(char))
```

The last special request is the list request. For this request, we just send a list of all the timestamps for the backup folders in existence. To do this, we use a for loop to go through the directory twice. The first time just calculates the length of the body by getting the length of the timestamps with a newline attached. We then send this header back to the client. The second run sends the actual timestamps to the client. In order to calculate all this, we first open the current directory. We check all directories that have “backup-” in the name. From there, we obtain the timestamp using sscanf() and then attach a newline using sprintf(). We can obtain the length of this string and add this number to the length variable with each iteration. This length is sent if it is the first iteration of the for loop. The timestamp is sent if it is the second iteration of the for loop.

```
for (int i = 0; i < 2; i++)
    int length = 0;
    opendir()
    while (readdir() != NULL)
        char timestamp[20];
        if (it is a directory && strstr(d_name, “backup-”) != NULL)
            sscanf(d_name, “backup-%s”, timestamp);
            sprintf(timestamp, “%s\n”, timestamp);
            length += strlen(timestamp);
            if (i == 1)
                send(commfd, timestamp, strlen(timestamp), 0);
    if (i == 0)
        sendheader(commfd, response, 200, length);
```

For a regular GET, we utilize `stat()` to get file size, `open()` the file, then append one character at a time to a larger string for file content, much like a variation of the structure of getting the HTTP request header. Then we call `sendheader()` to write a 201 Created HTTP response. We also need to check if the requested file does not exist (404 File Not Found) or if we don't have reading permissions to the file (403 Forbidden) and call `sendheader()` accordingly to write the appropriate HTTP response.

6. Handle PUT Request

If the client sends a PUT request, the first thing done is to check for the content length if it is provided. This tells the server how much of the body we want to write to a file. If the content length isn’t provided, the entire body is written to the file. In order to implement this, an if-else statement is used. If the length is provided, a for loop is used to `recv()` the body and store it in a string array until that content length is hit. Otherwise, we `recv()` until the client closes the connection to signal that they have nothing else to send.

Once the body is obtained, we use `open(filename, flags, permissions)`. The flags we include are `O_CREAT, O_WRONLY,` and `O_TRUNC` and they are bitwise ORed. The `O_CREAT` flag creates a new file if the file name doesn’t exist yet. `O_WRONLY` provides write access and `O_TRUNC` ensures that the file is overwritten if it exists by truncating the length to 0.

The permissions for the file are `S_IRUSR, S_IWUSR, S_IRGRP,` and `S_IROTH. S_IRUSR` and `S_IWUSR` give the user read and write permissions. `S_IRGP` and `S_IROTH` gives the group and others read permissions.

```
putfd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR |
      S_IRGRP | S_IROTH);
```

Once the file is opened, we can write to the file with the content in the char array and then close the file. At the end we call `sendheader()` with the appropriate parameters.

At the end of our infinite while loop, we just `memset()` all of our char arrays to 0 and close the connection.

Functions:

- Converting hostname and IP address: `getaddr()`
  We also use a function our TA Daniel Santos Ferreira Alves provided:

  ```
  /*
  getaddr returns the numerical representation of the address identified
  by *name* as required for an IPv4 address represented in a struct
  sockaddr_in
  Input: string of hostname or ip address
  Output: unsigned long of address
  */
  unsigned long getaddr(char *name);
  ```

  This function encapsulates `getaddrinfo()` which converts text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures (a data structure used to represent addresses and hostnames within the networking API).

- HTTP Status Codes: `getStatus()`
  Two functions are created in order to handle the HTTP status codes and to send the response back to the client. The first function `const char* getStatus(int code)` is used to enumerate the status codes to string literals. This is implemented using a switch statement with several cases that signify the `200 OK`, `201 Created`, `400 Bad Request`, `403 Forbidden`, `404 Not Found`, and `500 Internal Service Error` status codes. The function takes an integer status code number and returns the corresponding status string. These status codes are used when we send the response back through the `sendheader()` function.

- Sending the Response: `sendheader()`
  The `sendheader()` function takes in the connection file descriptor, the char array storing the response, the status code, the content length, and the body of the response if there is one.

  ```
  /*
  This function formats a HTTP response using sprintf() then send()
  response to client
  Inputs:
  - commfd <- port # for accept() connection
  - response <- pointer to store HTTP response string
  - code <- status code #
  - length <- length of body of response (not the header)
  - content <- the actual content; a pointer to string
  Output: void
  */
  sendheader(int commfd, char* response, int code, int length, char*
  content)
  ```

Testing:

We used a combination of unit testing and whole system testing. To check for errors and proper http status code response handling, we used unit testing because we are more concerned with functionality with errors here. As for checking to see if our server replicates the basic behaviors of a http server with a curl client, we used whole system testing because we want to see if our program just works. These are the testing steps we took:

Unit Testing

- check for correct number of arguments
- check if port number is set properly
- print full request header
- print parsed request type
- print parsed file name
- check for 400 Bad Request
    - check if request is HTTP/1.1
    - check if file is 10 characters
    - check if file is alphanumeric
- check for 500 Internal Service
    - check if using only GET and PUT HTTP methods (e.g. not HEAD, POST, etc)
- check for 404 File Not Found
    - If GET, check if file exists
- check for 403 Forbidden
    - If GET, check if file is readable
- If GET, print file content
- If PUT, print content length if provided & file content
- In terminal, `ls -l` to check file permissions: `rw-r--r--`

Whole System Testing

- GET / POST
    - `curl http://localhost:8080/file -v`
    - `curl localhost:8080/file[0-2] -v`
    - `curl -T input http://localhost:8080/output -v`
    - `curl -T input http://127.0.01:8080/output -v`
- Backup
    - `GET/b with only the binary in that folder 200/201`
    - `GET/b with many files in that folder (+30) 200/201`
    - `GET/b with half of the files that have permission,  half files with no permission 200/201`
    - `GET/b with all files having no permission 200/201`
    - `GET/b with no files (don't think this is possible but its an idea worth mentioning) 200/201`
- Recovery
    - `GET/r on valid recovery folder 200/201`
    - `GET/r where no recovery folder exists 404`
    - `GET/r on recovery folder with no permissions 403`
    - `GET/r on valid recovery folder but files inside have no permission 200/201`
    - `GET/r on an empty recovery folder (do we erase all the items in the server folder then?) 200/201`
    - `GET/r on a valid recovery folder where the files already exist in the server folder.  (overwrite the server folder files) 200/201`

    - GET/r where the files in the recovery folder have full permissions but the same files already exist in the server folder and have no permission 200 - files in server folder not overwritten
- Specific Recovery Folder
    - GET/r/backup-number on valid recovery folder 200/201
    - GET/r/backup-number on recovery folder that doesn't exist 404
    - GET/r/backup-number on recovery folder with no permissions 403
    - GET/r/ on valid recovery folder but files inside have no permission 200/201
    - GET/r/backup-number where the recovery folder is empty (do we erase all the items in the server folder then?) 200/201
    - GET/r/ with no backup folder specified 400
    - GET/r/backup-number on a valid recovery folder where the files already exist in the server folder. (overwrite the server folder files) 200/201
    - GET/r/ where the files in the recovery folder have full permissions but the same files already exist in the server folder and have no permission 200/201
- List
    - GET/l on a single backup folder 200
    - GET/l on many backup folders (+30) 200
    - GET/l where no backup exists 200
    - GET/l on a many backup folders that have no permission 200

Assignment Question:
Q: How would this backup/recovery functionality be useful in real-world scenarios?
A: Backup and recovery can be extremely useful when files get corrupted or are accidentally deleted. Having backups from previous times allows users to handle these problems efficiently and reliably. This is also beneficial for companies to protect them against data loss. Having multiple backups also allows us to restore back at a point in time. A real-world example can be using git to push multiple versions of a file to a repository. If the file has been corrupted or lost, we can git pull to get the most recent commit. If the file is a source code file that is in development and no longer compiles or keeps getting a segmentation fault, the developer can find a specific commit at a point in time when the source code file was running properly and respond appropriately to debug and fix their current errors or start fresh from that commit.