Asgn2 Design Document

Prepared by: Shirley Phuong & Miranda Eng
CSE130 Principles of Computer Systems Design
November 30, 2020

Program Description:
This program implements a multi-threaded HTTP server that responds to GET and PUT requests. In order to choose the number of threads, the user can run the code with the -N flag followed by the number of threads. Users also have the option to implement redundancy in the server by providing the -r flag. The code uses the POSIX thread library to create a thread pool of worker threads. The worker threads perform the request handed to them by the single dispatcher thread. If the -r flag is provided, the server performs GET and PUT requests in the copy1, copy2, and copy3 folders. GET requests compare all of the files before returning the contents. PUT requests write to each of the copies. The foundation of the GET and PUT methods is the same as Assignment 1, requiring file names of 10 ASCII characters. The GET method just returns the contents in the file and the PUT method writes the body of the request into the file.

Design:
In order to pass information between the dispatcher and worker threads, we created a struct holding all the necessary variables and data structures. The main data structures we used were a queue to hold the connection file descriptor for each request and a vector that kept track of the files in the server and their mutexes.

Program Logic:

Main Function:
1. Set default value for number of threads and for the redundancy flag
2. Check command line arguments
3. Setup TCP connection
4. Set shared data struct
5. Initialize lock for each file in copy directories
6. Create pthreads
7. Create dispatcher
    a. Wait for worker thread to consume if session queue is full
    b. Accept a connection from client
    c. Add this session to session queue

Worker Function:
1. Set pointer to shared data -
2. Lock global mutex and block until there are requests in the queue -
3. Get HTTP request -
4. Parse HTTP request header -
    a. check for errors
5. Set a pointer to the `file_data` struct if it exists -
6. Handle GET request -
    a. section for GET without redundancy
    b. section for GET with redundancy
7. Handle PUT request
    a. section for PUT without redundancy
    b. section for PUT with redundancy

Helper functions: `getStatus(), long getaddr(), sendheader()`

Structures: `file_data, sessions, shared_data`

__Main Function__:

1. **Set Default Value for Number of Threads and for the Redundancy Flag**
   We set the default number of worker threads to 4 and the redundancy flag to 0.

2. **Check Command Line Arguments**
   We use `getopt()` which parses the command line options: `N` & `r`. If the user specifies `-N x`, then set the number of worker threads to `x`. If the user specifies `-r`, then set the redundancy flag to `1`. In order to implement this, we use a while loop to go through the arguments. Within this while loop, we use a switch statement to filter for the flags.

3. **Setup TCP Connection**
   We follow the same familiar TCP server-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.

4. **Set Shared Data**
   We declare our data type `shared_data` so all threads can access shared variables safely. Initialize global, dispatcher, and worker mutexes. Then set the corresponding values (rflag, sockfd, numthreads, currentrequests, servaddr) to in variables in our `shared_data` struct.

5. **Initialize lock for each file in copy directories**
   1) Open the main directory (./). If the redundancy flag is 0 (no redundancy), then traverse through the directory for files and create a `file_data` struct for each file.
   2) If the redundancy flag is 1 (have redundancy), then open copy1 directory and traverse through the directory for files. Create a `file_data` struct for each file. For the copy2 and copy3 directories, also check if the file already has an element in the vector (by checking the file names).

6. **Create pthreads**
   Use a for loop to create the number of worker threads that we want. Do this using
   ```
   pthread_create().
   pthread_t worker_tid[SIZE];
   for (int i = 0; i < num_workers; ++i) {
         pthread_create(&worker_tid[i], NULL, &worker, &common_data);
   }
   ```

7. **Create Dispatcher**
   The rest of the main function is an infinite while loop that acts as the dispatcher thread. We use the global lock to block this thread if there is no space to accept new connections.

   ```
   pthread_mutex_lock(&common_data.global_mutex);
   while (common_data.currentrequests >= common_data.numthreads) {
         pthread_cond_wait(&common_data.dispatcher_cond,
               &common_data.global_mutex);
   }
   pthread_mutex_unlock(&common_data.global_mutex);
   ```

   Once there is space for a new request, it accepts the new connection and uses the global lock again before pushing the new request to the queue for the worker threads. At this point, it signals to any blocked worker threads that there is a new request.

Worker Function:

1. **Set Pointer to Shared Data**
   We start off by creating a `struct shared_data` pointer. This will allow us to alter variables in the struct and have it apply to all threads.
   `struct shared_data* shared = (struct shared_data*) data;`

2. **Lock Global Mutex While Request Queue is Empty**
   We have to block the threads if there are no requests to perform. In order to do this, we lock the global mutex and use a while loop. If the queue is empty, we wait using the `pthread_cond_wait()` function. This function takes in a condition variable for the worker and the global mutex. It unlocks the mutex and waits for a signal to the condition variable before it locks the mutex again. Once a new request arrives in the queue, we take the connection file descriptor from the queue and pop the first element from the queue. Once this is done we can finally unlock the global mutex. This part of the code is a critical section.

```
pthread_mutex_lock(global_mutex);
while (request queue is empty) {
      pthread_cond_wait(worker_cond, global_mutex);
}
commfd = requestqueue.front().commfd;
requestqueue.pop();
pthread_mutex_unlock(global_mutex);
```

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, " ");
fileName = 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. 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.
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. **Set a Pointer to the `file_data` Struct if it Exists**
After checking for all of the errors we can create a `struct file_data` pointer. We use a for loop to iterate through all of the elements in the `file_data` vector and compare the request's file name to all of the file names in the vector. If we find an element with the same name, we set the pointer to that struct. This pointer will allow us to lock the file mutexes when we perform the GET and PUT requests.

```
struct file_data* filepointer;
int vectornum = 0;
for (iterator i = file_data.begin(); i != file_data.end(); ++i) {
        if (file names are the same) {
                filepointer = (struct file_data*) &file_data[vectornum];
                break;
        }
        vectornum++;
}
```

6. **Handle GET Requests**
There are two different GET requests that our server can perform. The first is the usual GET request with no redundancy. This part of the code will be taken from our Assignment 1 program. The only difference is that we added file locks when we perform the actual request. We start off by checking if the file exists (404 File Not Found) or if the file has reading access (403 Forbidden). We use `stat()` to get the size of the file and use this number to send the header using `sendheader()`. We then use a while loop to `read()` the file contents and `send()` to send the contents character by character to the client.

The second type of GET request is one with redundancy. This version is a lot more complicated because we have to compare 3 versions of the files and use the version that is part of the majority. The first thing we need to do is create paths to every copy by using `sprintf()` and inputting a path template as well as the file name.

```
sprintf(path, "copy1/%s", filename);
```

In order to error check, two 3-element arrays are used. They are used as error flags for reading access and file existence. After this we can check if a majority of the files have errors.

```
int majorityF[3] = { 0 };           // errors for file not found
int majorityR[3] = { 0 };           // errors for no reading access
```

Once we set these flags we can use these as parameters for the various error checking if-else statements. Depending on which copy has these errors we will compare the other two copies. If all three copies have no errors we can compare all three of them.

This checks for if most copies have errors.
```
if((majorityF[0] + majorityF[1] + majorityF[2]) >= 2) {
      send error response
}
```

This singles out a copy with errors to compare the other two.
```
if ((majorityF[0] == 1) || (majorityR[0] == 1)) {
      compare copies 2 and 3
}
```

In order to compare two files, we use a while loop to read each file character by character into two different buffers. From there, we can use `memcpy()` to compare the two buffers. This function takes in both of the buffers as well as the size we want to compare. If it returns 0 through every iteration of the while loop, we know that the two files are the same. If the files are not equal we repeat everything but move on to a different set of two files.

```
while ((read(getfd1, buf1, sizeof(char)))) {
      read(getfd2, buf2, sizeof(char));
      if (memcmp(buf1, buf2, sizeof(char)) != 0) {
            errors = ERROR;
            break;
      }
      memset(&buf1, 0, sizeof(buf1));
      memset(&buf2, 0, sizeof(buf2));
}
```

If we find at least 2 of the copies are the same, we can finally send it to the client. We set the official path to the path of one of the correct copies and obtain the file size using `stat()`. From there we can send the data similar to how we sent it with no redundancy. We `open()` the file and `read()` the data using a while loop. We then `send()` character by character to the client.

7. **Handle PUT Requests**
Similar to the GET request, there are two different versions of the PUT request- one implementing redundancy and one without. For the most part, both versions are pretty similar. However, a PUT with redundancy has to write to all three copies of the file.

The PUT request is split up into four different parts. The first part is for PUT requests where the content length is specified and the file did not exist previous. The second part is where the content length is specified but the file already exists which means we have to overwrite the data. The last two parts are similar but the content length is not specified for both.

For the parts that have a file that did not exist previously, we have to create a new `file_data` struct and push it to the vector. Before doing this we also have to lock the global mutex in case any other requests also want to create the same file, making this a critical section.

```
pthread_mutex_lock(global_mutex);
struct file_data file_data;
strcpy(file_data.filename, filename);
pthread_mutex_init(file_mutex, NULL);
```

```
shared->fdata.push_back(file_data);
filepointer = (struct file_data*) file_data.back();
pthread_mutex_lock(file_mutex);
// unlock after request is performed
```

After this we can open each of the copies and write to them. For a PUT without redundancy this is the same as the previous assignment. In order to implement redundancy, we use a for loop to put all of the file descriptors in an array while also opening the file. 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.

```
char path[100];
for (int d = 0; d < 3; d++) {
      sprintf(path, "copy%d/%s", d+1, filename);
      putfd[d] = open(path, O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR
            S_IWUSR | S_IRGRP | S_IROTH);
}
```

After this we use a for loop to to `recv()` the body of the request and write to the copies. Inside of this for loop, we use another for loop to write to all three copies.

```
for (int i = contentlength; i > 0; i--) {
      n = recv(commfd, buf, sizeof(char), 0);
      write(STDOUT_FILENO, buf, sizeof(char));
      for(int j = 0; j < 3; j++) {
            n = write(putfdr[j], buf, sizeof(char));
            if(n < 0) { err(1, "write()"); }
      }
      memset(&buf, 0, sizeof(buf));
}
```

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 and unlock the mutexes.

Overall, this program is thread-safe because of the use of the global mutexes and the file mutexes. The use of the global mutexes forces other threads to stop if important shared variables are being altered. In this case, the mutex has to be locked when there is a PUT request with a new file. In addition to this, the file mutexes don't allow race conditions to occur. The locking of specific files will only allow one request pertaining to that file to be processed while the others have to wait until this section is over.

Helper 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)
```

<u>Structures</u>:

-
  ```
  // This struct safely holds data that all threads can access
  struct shared_data {
        pthread_mutex_t global_mutex;
        pthread_cond_t dispatcher_cond, worker_cond;
        int rflag;
        int sockfd;
        int numthreads;
        int currentrequests;
        int fileexists;
        struct sockaddr_in servaddr;
        std::vector<file_data> fdata;
        std::queue<sessions> session_queue;
  };
  ```
  - **global_mutex**: a global lock that is used for critical sections such as accepting new connections and creating a new file for PUT requests

    - **dispatcher_cond**: condition variable used to signal to the dispatcher function when there is space to accept new requests
    - **worker_cond**: condition variable used to signal to the worker function when there is a new request to be processed
    - **rflag**: integer variable that is set when the -r flag is given
    - **sockfd**: holds the fd of the socket for connections
    - **numthreads**: holds the number of threads that the user input in the command line argument; if no flag is given, it is set to the default value of 4
    - **fileexists**: flag that is set when the worker function is searching for the correct file_data struct
    - **servaddr**: variable for socket binding
    - **fdata**: a vector that holds elements of the `file_data` struct, elements are initialized in the `main()` function and files can be added on using PUT requests
    - **session_queue**: a queue that holds elements of the `sessions` struct, elements are added whenever the dispatcher thread accepts a new connection and popped when the worker thread takes in a new request

-
  ```
  // This struct holds the connection file descriptor for one request
  struct sessions {
      int commfd;
  };
  ```
    - **commfd**: communication file descriptor for a specific request

-
  ```
  // This struct holds the file name and mutex for one file
  struct file_data {
      char filename[100];
      pthread_mutex_t file_mutex;
  };
  ```
    - **filename[100]**: char array holding the file name for a file
    - **file_mutex**: the mutex for the corresponding file, used to lock in the worker thread when the file is used for a request

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 flags are being accounted for
- check if port number is set properly
- print all files in the file_data vector (for both redundancy and no redundancy)
- 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:
- `curl http://localhost:8080/file1 -v & curl http://localhost:8080/file2`
- `curl localhost:8080/file[0-2] -v & curl -T t1 localhost:8080/file`
- `curl -T input http://localhost:8080/output -v & curl -T input2 http://localhost:8080/output`
- `curl -T input http://127.0.01:8080/output -v & curl -T input http://127.0.01:8080/output2 & curl http://localhost:8080/binaryfile`

<u>Assignment Questions</u>:
Q: If we do not hold a global lock when creating a new file, what kind of synchronization problem can occur? Describe a scenario of the problem.
A: If we do not have global lock when creating a new file, multiple requests can try to create that same file. Since there is no file lock for this file, they can end up both writing to the same file which will result in synchronization problems. Instead of the file containing data from only one request, it would be a mix of both. In addition to this, multiple mutexes could be created for the same file which can result in more synchronization problems down the line if the same file mutex is not being locked.

Q: As you increase the number of threads, do you keep getting better performance/scalability indefinitely? Explain why or why not.
A: Performance can be latency or throughput, Here, we will assume performance is throughput, the number of requests per second. Multithreading improves performance by allowing multiple threads to work at the same time, however, it also introduces additional overhead (i.e. locks, signals, etc). Multithreading is good, but at some point too much multithreading would plateau performance because there might just not be enough work for each worker to do. So some workers are just stuck queueing behind another worker all the time, clogging up the queue, and not speeding up activity. You want to have enough workers so that every worker can work without having to queue after one another. After reaching this point, increasing threads degrades performance.
Scalability is the ability to process more requests per second as the number of threads grows. However, the scalability would not get better indefinitely because there is only one dispatcher thread. The number of requests per second or performance would be bottlenecked by the speed the dispatcher thread can receive requests and pass them to the worker threads.