Design Document:
alias_multi-threaded_httpserver

Arsh Malhotra
Cruz ID: amalhot3

1 Goals

The goal of this program is to create an sustain an HTTP server that can take GET and PUT requests and output the desired response.

2 Design

This program required handling arguments, setting up the server, handling a GET request and PUT request and sending their response. Additionally, to handling the requests, is to properly read in the headers and content, if any. On top of these, this assignment adds a multi-threading and synchronization requirement along with logging. Finally, we add aliases to the mix of logic.

2.1 Handling arguments

This is a simple task and requires checking for arguments and making sure there is at least a hostname and my logic is displayed in Algorithm 1.

```
   Input  : Argument count: argc
   Input  : Argument list: args[0..argc]
   Global : Number of threads to create: nthreads
1  if argc < 2 then
2  |  throw error
3  hostname -> args[1]
4  port -> "80"
5  if argc > 2 then
6  |  port -> args[2]
7  end
8  opt <- getopt(argc, argv, "N:")
9  while opt != -1 do
10 |  switch opt do
11 |  |  case 'N':
12 |  |  |  nthreads <- atoi(optarg)
13 |  |  end
14 |  end
15 end
```

Algorithm 1: No argument code.

2.2 Setting up the server

I used the code a TA had provided on Piazza to create the socket. I implemented error handling myself for the code for the methods `socket`, `setsockopt`, `bind`, `listen`, and `accept`. The error handling depended on each method's error code.

2.3 GET & PUT

Both GET and PUT require similar handling. The key difference is that PUT comes with an attachment and GET responds with one. Reading the header for each is a simple task as I read in from the socket using `read(2)` 1 byte at a time so that I can identify where the line break occurs to fully identify where the header is. Once the header is acquired I will identify the first three characters to determine whether it's a PUT or GET request. If it's a GET request, I will then evaluate the resource name to check whether it is valid and then attempt to retrieve the contents of the desired resource. After retrieving the contents I will write back to the socket with the contents. If the contents couldn't be found (the file was not found) I would throw a different error appropriate to that (404). If a PUT request is found, I will also check for a valid resource name and then proceed to check for the `Content-Length` flag in the header. If found then, I will use `read(2)` to only read the length that is specified by said flag. If not found, then I will use `read(2)` to keep reading until EOF. Afterwards I will write back to the socket if successful or not.

```
  Global : Int—max size to read: max_size (default = 16000)
1 function read_from_socket(socket):
2 |  size <- read(STDIN, buffer, max_size)
3 |  while size > 0 do
4 |  |  header += buffer
5 |  |  size <- read(STDIN, buffer, max_size)
6 |  end
7 |  request <- header.substring(0,3)
8 |  if request == "GET" do
9 |  |  handle_get(header, socket)
10|  else if request == "PUT" do
11|  |  handle_put(header, socket)
12|  else
13|  |  send_bad_request()
14|  end
```

Algorithm 2: Code to read from the socket.

```
  Global : Int – max size to read: max_size (default = 16000)
1 function handle_get(header, socket):
2 |  resource <- check_resource(header)
3 |  if resource is valid do
4 |  |  fd <- open(resource)
5 |  |  if open failed do check_fail_and_return_response()
7 |  |  else
8 |  |  |  write_header_to_socket(socket)
9 |  |  |  size <- read(fd, buffer, max_size)
10|  |  |  while size > 0 do
11|  |  |  |  write(socket, buffer, size)
12|  |  |  |  size <- read(socket, buffer, max_size)
13|  |  |  end
14|  |  end
15|  else
16|  |  send_bad_request()
17|  end
```

Algorithm 2: Code to handle GET request.

```
  Global : Int – max size to read: max_size (default = 16000)
1 function handle_put(header, socket):
2 |  resource <- check_resource(header)
3 |  if resource is valid do
4 |  |  fd <- open(resource)
5 |  |  if open failed do check_fail_and_return_response()
7 |  |  else
8 |  |  |  content_length <- find_content_length()
9 |  |  |  i <- 0
10|  |  |  while content_length > max_size * i do
11|  |  |  |  read(socket, buffer, max_size)
12|  |  |  |  write(fd, buffer, max_size)
13|  |  |  end
14|  |  |  rest_to_read <- content_length – (max_size * i)
15|  |  |  read(socket, buffer, rest_to_read)
16|  |  |  write(fd, buffer, rest_to_read)
17|  |  end
18|  else
19|  |  send_bad_request()
20|  end
```

Algorithm 3: Code to handle PUT request.

2.4 Multi-threading & Synchronization

For multi-threading and synchronization, I went with the producer-consumer model. The main thread is my producer and I create threads for my consumers who will be consuming the accepted socket. The consumer would call the `read_from_socket` function from Algorithm 2.

```
int32_t nthreads;
Semaphore empty(n), full(0), mutex(1);
int32_t socketBuffer[n];
int32_t out <- 0;
```

Producer

```
int32_t in <- 0;
int32_t socket;
while True {
  socket <- accept(socket_fd);
  if socket < 0 then
    throw error
  empty.down();
  mutex.down();
  buffer[in] <- socket;
  in <- (in+1) % nthreads;
  mutex.up();
  full.up();
}
```

Consumer

```
int32_t socket;
while True {
  full.down();
  mutex.down();
  socket <- sockBuffer[out]
  out <- (out+1) % nthreads;
  mutex.up();
  empty.up();
  read_from_socket(socket);
}
```

Algorithm 4: Producer-Consumer modified slightly.

2.5 Logging

I had to code this while writing this because I couldn't understand the logic I was trying to portray, so for Logging design I only have text. For logging, I decided to create a global variable `offset` which also meant that I would have to use Semaphores. I created a Semaphore `offsetMutex` so that I can control and synchronize the changing of `offset`. I calculate the space needed to be reserved for logging before I access `offset` by taking the fixed lengths given in the assignment description and adding the variables to it.

```
reserve = 39 + length_of_string_content_length + 69 *
          floor(content_length/20)
if (content_length % 20) do reserve += (9 + 3*(content_length%20))
```

I would then `offsetMutex.down()` and get the current offset and store it into a local variable and then add `reserve` to `offset` and `offsetMutex.up()` for another thread to access it. This way I have reserved the space necessary and have found the where to write to. Both PUT and GET requests have their own variation for writing to `log_file` but the main logic behind that was to take in a buffer and take 20 characters at a time to build each `line` which would then be `pwrite(log_file, line, 69, threadOffset, bufferOffset)` until there were 20 or less characters left. At that point I would create `line` dynamically as before and find the length of `line` and `pwrite(log_file, line, lenLine, threadOffset, bufferOffset)`.

```
writeToLog(char_num, buffer, thread_offset, buffer_offset,
read_length):
    for (j=0; j<8-length_line_number; j++) do
        strcat(line, zero);
    end
    strcat(line, line_number);
    strncpy(buf, buffer+buffer_offset, read_length);
    for (j=0; j<read_length; j++) do
        sprintf(hex, " %02x", buf[j]);
        strcat(line, hex);
    end
    strcat(line, nl);
    pwrite(log_file, line, strlen(line), threadOffset);
}
```

Algorithm 5: Writing successful response to log file.

```
writeFailToLog(request, resource, respCode):
    failLineLength = 37 + len(resource);
    sprintf(failLine, "FAIL: %s %s HTTP/1.1 --- response %s\n",
request, resource, respCode);
    offsetMutex.down();
    threadOffset = offset;
    offset += failLineLength;
    offsetMutex.up();
    pwrite(log_file, failLine, failLineLength, threadOffset);
}
```

Algorithm 6: Writing failed response to log file.

2.6 Aliases

I plan making another option and using the argument to open a file either as create or read/write. If the file is empty/just created then I will add to the first line a magic number that I have randomly generated:

```
BR6W7PPO9Y6U5IJFSUYLRURZISHE7CICMUBAV573OHELBBQZYRNBRTEJYG21MLL6
```

I will then scan for the first open line and set that as the offset. The offset will be a variable only edited within a critical section. It will be incremented by 128 or until a NULL terminator is found (128-character increment would be first priority). At an empty 128-character location, the key-value pair will be printed as "<new_name>:<existing_name>".

For PUT and GET requests, I'm going to check for whether the file exists first if it's a valid resource name, and if that fails then check if it's in the mapping file as a key. All other cases, I will check for it as a key in the mapping file.

DISCLAIMER: I didn't really understand what needed to be done until I started coding so I also don't know what pseudocode is required.