Asgn3 Design Document

Jessica Pan

CruzID: jeypan

CSE130, Fall 2019

1. Goal

The goal of this assignment is to modify a previous HTTP server implemented in Asgn1 into a server that has the additional features: caching and logging. The feature of caching in the HTTP server is to maintain a buffer in my server that contains a subset of the pages, in our case, 4 pages. When a request is received, if the requested page is in the cache, then it's read from the cache (GET) or updated in the cache (PUT). Otherwise, the page is first read from disk into the cache. As for the logging feature, it'll be the same as the logging feature implemented in Asgn2. However, this time, when logging, the files will indicate whether the page was in the cache at the time the request was received.

2. Assumptions

The assumptions below are made from Asgn1 and they will also be applied to this assignment:

The assumptions I'm making for this program is when a GET request is made, the server will understand the request and print the data corresponding to the file name specified in the request (assuming that the file name satisfies the file name criteria). I'm also assuming, since clients, a web browser, making GET requests from HTTP servers don't store the pages received into their own directory, therefore, my HTTP server will not store the file requested onto the client's directory. For a GET request, the server will simply write the data to the client's standard output.

Next is when client sends the PUT request, if the file isn't able to be accessible by curl or if the file isn't present in the client's directory for curl to call it to PUT into the server, then curl would, on its own, disconnect the client-server connection with an error of curl: can't open 'filename' when it's ran with the command: curl --upload-file filename [ip address]:[port #] or curl –T filename [IP address]:[port #].

Another assumption I'll be making is that when the server is binded to an IP address, the client must call the same IP address and the same port to the server in order to make the proper connection to the server. For example, if the server were to set it's IP address to 127.0.0.2 with port 8080, then the client must call the same IP address and port number: curl 127.0.0.2:8080.

Assumptions made during this assignment:

Since we're required to log a file with the indication of whether or not it was in the cache when it was requested, the '[was in the cache]' and '[was not in the cache]' will be logged only for status codes of 200 (GET/PUT), and 201 (PUT). Because the requests with status codes 400, 403, 404 and 500 means that the server can't perform the request or the client doesn't have access to the files. Thus, it's not possible for a page to be stored in the cache when there's no accessibility to the page's contents to perform the request.

I'm also assuming that we don't have to handle the case when the server's cache contains files, and when the server is turned off the server would write all the files in the cache to disk as I was informed on piazza that I don't have to handle that case.

3. Design

The approach I'm taking to this program is to first check the arguments when the user starts to run the program. From the arguments, there must be an IP address specified and an optional port number. In addition to the IP address and port number, the user will also get the choice to turn on the caching flag (-c) and logging flag (-l [logfile name]). I used `getopt()` to get the flag values inputted by the user to set the server up to perform caching and/or logging when the flags are turned on.

If the port number isn't specified, then the standard port number will be set, which is 80. After setting the port number to a variable, the socket file descriptor is created. If program is unable to create socket file descriptor, program will exit with exit failure. Once the socket file descriptor is created, the next task was to define the IP address and port number specified in the arguments when the program was ran. As we define the IP address, we need to make sure to check if the user used an alias for the IP address (i.e. localhost). If the user did use an alias for the IP address, in this case "localhost," then the program will convert it to "`127.0.0.1`" and set that as the IP address.

After the IP address and port number is set, the program will bind the address to the server using the function, `bind()`. The server will then listen for the socket using the function, `listen()`. If these two functions return -1, then the program will exit with EXIT_FAILURE. If not, then the program will create a connection. To maintain this connection, for the server to be constantly listening, there will be a while loop that'll always be true, `while(1)`.

Once the program is in the while loop, the program will wait for a connection and accept a connection with the client with the function `accept()`. This function will only accept clients specifying the same IP address and port number as the server in its GET/PUT requests. If this function returns a -1, then the program will exit with EXIT_FAILURE. Otherwise, the program will continue to parse the GET/PUT header received from a client.

The program will create a logfile of [*logfile name*] specified by the user if they do decide to turn on the logging feature and the program will then parse the header of the request with the function `strtok()` with the delimiter being "`\r\n`." By doing this, it'll place each line in the header into a char array of characters. Then, the program will parse each line with the same function but with a space delimiter. The program will then `push_back()` the values of the header to its corresponding key in a vector of strings.

After storing all the values of the header, the program will then begin to check whether the file specified by the client is valid. For the file name to be valid, it needs to be exactly 27-ASCII characters, but these characters must consist of the characters '`A`' to '`Z`', '`a`' to '`z`', '`–`', and '`_`'. Any other characters that don't fall within this range will cause the program to send the client a status code response of 400 (Bad Request) and then the server will wait for another connection from another client.

If the file name fits the specified criteria, then the program will check what method is used for the client's request. If their request is not a GET or PUT, then the program would respond to the client with a status code of 500 (Internal Server Error). The server would then wait for the next connection made from a client.

If the method is a GET and the file name is valid, then the program will open this server with the function `open()`. Depending on the value returned by `open()`, if it's -1, then the program will send a 403 status code (for accessing a forbidden file on the server) or a 404 status code (if the file doesn't exist on the server). If the returned value is not -1, then the server will `send()` a 200 OK status code and the content length of the file, also it'll `write()` the data of the file to the client's standard output after reading the data with `read()`. This GET method does have one exception. That is if there's no file name specified by the client, then the server will write no data to the client's standard output and `send()` a 200 OK status code because the client isn't requesting any data. Once this is done, the program will wait for the next connection and request from a client.

If the client does turn on the caching feature for the server, then at the time of a GET request, the server will loop through the cache. The cache in the server will be a deque structure that stores pairs, the pair being the file name and the file's contents. The loop will compare the file name of the GET request to the first element (file name) of each pair in the deque. If the file names are the same, the data for the file requested in the GET request will be returned to the client from the cache. However, if the cache doesn't hold the file that's being requested, the data of the file being requested will be read from the disk and those contents will be stored into the cache and also be returned to the client. As long as the cache isn't full (< 4 pages with different names) then the client will be able to *push_back()* pages to the cache (the deque structure) without evicting pages. If the cache is full (== 4 pages) then the cache will evict the oldest page stored in the deque and also create that file on the disk and write the contents into the file.

As for logging, if the cache feature is turned on, the header of GET will be logged with '[was in cache]' if the file was in the cache when the GET request was performed. If the file wasn't in the cache when the GET request was performed, the GET header will be logged with '[was not in cache]'. If the cache feature isn't turned on or the status codes to the requests made are 400, 403 and 404, the GET header will just be logged by itself.

If the method is a PUT and the file name is valid, then the program will then check if the file being placed in the server is forbidden (i.e. if the client has permission of the file to place into the server). If it's forbidden then a status code of 403 will be sent to the client. If the file that the client is trying to put to the server doesn't exist in the client's directory then my program would return a 400-status code. Except if the client were to put a file to the server that didn't exist in the client's directory `curl()` would stop the client from making this request.

Once the file being put is valid, the program will check if the content length in the PUT header was specified. The results depend on the specification of the content length. If the content length isn't specified, then the server will send the client a status code of 201 (Created) and continue to read the data from client and write the data to a newly created file on the server's directory. The server will continue to `read()` until the client closes the connection by typing *ctrl+c* in the client's terminal. If the content length is specified then the program will send a 201 (Created) status code and write the data of content length into the newly created file into the server's directory. The status code of 201 will be sent to the client if the file never existed on the server's directory before the PUT request, but if the file already exists on the server's directory, a status code of 200 OK will be sent back to the client.

If the client turns on caching for this server, then at the time of a PUT request the data of the page sent from the client will be stored into the cache first and not to disk. If the file name corresponding to the PUT request is already in the cache then the new PUT request's contents will overwrite the data of the file already in the cache. If the file doesn't exist on the cache, then the PUT request will push its data onto the cache. However, if the cache is full at the time of a PUT request, then the file name will be compared to the files in the cache. If they have the same name then the current PUT request will write over the file in the cache. But if the file isn't in the cache and the cache is full, the cache will have to remove its oldest file, and write its contents to the disk before the new PUT request can be written to the cache. Thus, if a PUT request is performed, it will write to the cache first, and only until the page is evicted from the cache, it will then write its contents to the disk. The logging feature for PUT is the same as a GET request where the header is logged and if caching is turned on, it'll also have the [was in cache] and [was not in cache] in the log file. However, the only thing that's different in logging for PUT is the file's data will also be written to the log file (in hex).

The very end, before the while loop ends, the program will close the socket connection between the server and client and then create a new socket once the while loop loops back around for the next connection.

4. Pseudocode

Below is the pseudocode for the multithreading httpserver with logging program.

```
Function parse_requests (int argc, char* argv[]) do
  if argc <= 6 then
    while (opt ← GETOPT(argc, argv, "cl:")) != -1 do
        switch(opt)
          case 'c':
            cache = true
            break
          case 'l':
            logfile.first = true
            logfile.second = optarg
       break
    end while

    if threadpoolSize == 0 then
         threadpoolSize = 4
    end if

    nonflags = argc - optind
    if nonflags == 2 then
        port = atoi(argv[optind + 1])
    else if nonflags == 1 then
        port = 80
    else
        WARNX("Usage: ./httpserver -N [# of threads] -l [log_filename] [address] [optional
port]")
        EXIT(1)
    end if
  else
       WARNX("Usage: ./httpserver -N [# of threads] -l [log_filename] [address] [optional
port]")
       EXIT(1)
  end if
  pair<int, int> port_and_cache
  port_and_cache.first ← port
  port_and_cache.second ← cache
```

```
    return port_and_cache
end parse_requests

Function handle_logging_offset (int newoffset) do
  prev ← offset
  offset += newoffset
  return prev
end handle_logging_offset

Function failed_request_log (string method, string fname, string protocol, string status) do
  return ("FAIL: " + method + " " + fname + " " + protocol + " --- response " + status +
"\n========\n")
end failed_request_log

Function successful_get_log (string method, string fname, bool cachingdone, bool cached) do
  if cachingdone then
    if cached then
      return (method + " " + fname + " length 0 [was in cache]\n========\n")
    else
      return (method + " " + fname + " length 0 [was not in cache]\n========\n")
    end if
  end if
  return (method + " " + fname + " length 0\n========\n")
end successful_get_log

Function successful_put_log (string method, string fname, string len, bool cachingdone, bool
cached) do
  if cachingdone then
    if cached then
      return (method + " " + fname + " length " + len + " [was in cache]\n ")
    else
      return (method + " " + fname + " length " + len + " [was not in cache]\n ")
    end if
  end if
  return (method + " " + fname + " length " + len + "\n")
end successful_put_log

Function handle_header_logging(string header, int o, int fd) do
  buffer[header.length() + 1]
  STRCPY(buffer, header.c_str())
  if logfile.first then
    k ← PWRITE(fd, &buffer, STRLEN(buffer), o)
  end if
end handle_header_logging
```

```
Function calculatetoReserve(int lines, int buffSize) do
   return ((8 * lines) + lines + 9 + (buffSize * 3))
end calculatetoReserve

Function handle_log_putdata (string header, char *datta, int count, int line, int off, int filedesc)
do
   tempOffset ← off
   head[header.length()]
   STRCPY(head, header.c_str())
   end[] ← "========\n"

   buff ← datta
   t ← PWRITE(filedesc, &head, SIZEOF(head), tempOffset)
   tempOffset += t

   linechars[61]
   previous ← 0
   padding ← 0
   for (int b = 0; b < count; ++b) do
      if (b != 0 && ((b % 20) == 0)) then
         temp ← TO_STRING(previous)
         temp ← STRING(8 - temp.length(), '0').APPEND(temp)
         pad[8]
         STRCPY(pad, temp.c_str())

         line[69]
         SPRINTF(line, "%s%s%c", pad, linechars, '\n')

         t ← PWRITE(filedesc, line, SIZEOF(line), tempOffset)
         tempOffset += t

         previous ← padding
      end if

      padding += 1
      SPRINTF(linechars + 3 * (b % 20), " %02x", buff[b])

      if b == (count - 1) do
         temp ← TO_STRING(previous)
         temp ← STRING(8 - temp.length(), '0').APPEND(temp)
         pad[8]
         STRCPY(pad, temp.c_str())
```

```
        leftover ← ((count % 20) * 3) + 8 + 1
        line[leftover]
        SPRINTF(line, "%s%s%c", pad, linechars, '\n')
        t ← PWRITE(filedesc, line, SIZEOF(line), tempOffset)
        tempOffset += t
      **end if**
    **end for loop**
    tempOffset += PWRITE(filedesc, end, STRLEN(end), tempOffset)
**end** handle_log_putdata

**Function** evictfileInCache()
    fname ← Cache.front().first
    temp ← Cache.front().second
    size ← temp.length()
       fdata ← to_string(Cache.front().second)

    Declare *fd* file descriptor
    **if** (fd ← OPEN(fname.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0666)) != -1 **then**
      WRITE(fd, fdata.C_STR(), fdata.SIZE())
    **end if**
       CLOSE(fd)
       Cache.pop_front()
**end** evictfileInCache

**Function** handle_connection(void new_socket, bool caching) **do**

    buffer[4000]
    *read_buf[4000]
    i ← 0
    status ← 0
    Declare *logfiledisc*
    **if** logfile.first **then**
      *logfiledisc* ← OPEN(logfile.second, O_RDWR | O_CREAT, 0666)
    **end if**
    READ(new_socket, buffer, SIZEOF(buffer))

    *token ← STRTOK(buffer, "\r\n")
    **while** token != NULL **do**
      read_buf[i++] ← token
      token ← STRTOK(NULL, "\r\n")
    **end while**

    **enum** HeaderKey { Method, Filename, Protocol, Host, Useragent, Accept, ContentLen,
Expect }
```

```
Create struct *S* of a char array *values*
Create vector of type *S* called *Headers*
tok ← STRTOK(read_buf[Method], " ")
**while** tok != NULL **do**
  **if** STRCMP(tok, "GET") == 0 or STRCMP(tok, "PUT") == 0 **then**
    s.values ← tok
    Headers.PUSH_BACK(s)
  **else if** STRCMP(tok, "HTTP/1.1") != 0 **then**
    s.values ← BASENAME(tok)
    Headers.PUSH_BACK(s)
  **else**
    s.values ← tok
    Headers.PUSH_BACK(s)
  **end if**
  tok ← STRTOK(NULL, " ")
**end while loop**

**if** STRCMP(Headers[Method].values, "GET") == 0 **then**
  t ← STRTOK(read_buf[1], " ")
  **while** t != NULL **do**
    **if** STRCMP(t, "Host:") != 0 **then**
      s.values ← t
      Headers.PUSH_BACK(s)
    **end if**
    t ← STRTOK(NULL, " ")
  **end while loop**

  t <- STRTOK(read_buf[2], " ")
  **while** t != NULL **then**
    **if** STRCMP(t, "User-Agent:") != 0 **then**
      s.values ← t
      Headers.PUSH_BACK(s)
    **then**
    t ← STRTOK(NULL, " ")
  **end while loop**

  t ← STRTOK(read_buf[3], " ")
  **while** t != NULL **do**
    **if** STRCMP(t, "Accept:") != 0 **then**
      s.values ← t
      Headers.PUSH_BACK(s)
    **end if**
    t ← STRTOK(NULL, " ")
  end while loop

else if STRCMP(Headers[Method].values, "PUT") == 0 then
  for (i = 1; read_buf[i] != NULL; ++i) do
    t ← STRTOK(read_buf[i], " ")
    while t != NULL do
      if (i == 1) && (STRCMP(t, "Host:") != 0) do
        s.values ← t
        Headers.PUSH_BACK(s)
      end if
      if (i == 2) && (STRCMP(t, "User-Agent:") != 0) do
        s.values ← t
        Headers.PUSH_BACK(s)
      end if
      if (i == 3) && (STRCMP(t, "Accept:") != 0) do
        s.values ← t
        Headers.PUSH_BACK(s)
      end if
      if (i == 4) && (STRCMP(t, "Expect:") == 0) do
        char eh[] = "None"
        s.values ← eh
        Headers.PUSH_BACK(s)
      else if (i == 4) && (STRCMP(t, "Content-Length:") != 0) do
        s.values ← t
        Headers.PUSH_BACK(s)
      else if (i == 5) && (STRCMP(t, "Expect:") != 0) do
        s.values ← t
        Headers.PUSH_BACK(s)
      end if
      t ← STRTOK(NULL, " ")
    end while loop
  end for loop
else
  status ← 500
  isrErr[] = "HTTP/1.1 500 Internal Server Error\r\nContent-Length: 0\r\n\r\n"
  SEND(new_socket, isrErr, STRLEN(isrErr), 0)
  if logfile.first do
    log ← FAILED_REQUEST_LOG(
     Headers[Method].values, Headers[Filename].values, Headers[Protocol].values, "500")
    off ← HANDLE_LOGGING_OFFSET(log.length())
    HANDLE_HEADER_LOGGING(log, off, logfiledisc)
  end if
end if
```

```
validFname ← false
fname ← Headers[Filename].values
if STRLEN(fname) != 27 then
   validFname ← false
else
   for (auto k = 0; fname[k] != '\0'; ++k) do
      if ((fname[k] >= 'A' && fname[k] <= 'Z') || (fname[k] >= 'a' && fname[k] <= 'z')
          || (fname[k] >= '0' && fname[k] <= '9') || (fname[k] == '-') || (fname[k] == '_'))
        && (fname[k] != '.')) then
         validFname ← true
      else
         validFname ← false
      end if
   end for loop
end if

if validFname == false && status != 500 do
   notTs[] ← "HTTP/1.1 400 Bad Request\r\nContent-Length: 0\r\n\r\n"
   SEND(new_socket, notTs, STRLEN(notTs), 0)
   if logfile.first then
      log ← FAILED_REQUEST_LOG(
        Headers[Method].values, Headers[Filename].values, Headers[Protocol].values, "400")
      off ← HANDLE_LOGGING_OFFSET(log.length())
      HANDLE_HEADER_LOGGING(log, off, logfiledisc)
   end if
end if

if (validFname == true) && (STRCMP(Headers[Method].values, "GET") == 0) do
   size_t nbytes, bytes_read
   buffy[4000]
   nbytes ← SIZEOF(buffy)
   filepath ← Headers[Filename].values

   if (pfd ← open(filepath, O_RDONLY, 0)) == -1 then
      Create struct stat s
      if stat(filepath, &s) == 0) then
         mode_t k ← s.st_mode
         if (k & S_IRUSR) == 0) then
            getErr[] ← "HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n"
            SEND(new_socket, getErr, STRLEN(getErr), 0)
            if logfile.first then
               log <- FAILED_REQUEST_LOG(Headers[Method].values,
                 Headers[Filename].values,
                 Headers[Protocol].values,
             "403")
            off ← HANDLE_LOGGING_OFFSET(log.length())
            HANDLE_HEADER_LOGGING(log, off, logfiledisc)
          end if
        end if
      else
        getErr[] ← "HTTP/1.1 404 Not Found\r\nContent-Length: 0\r\n\r\n"
        SEND(new_socket, getErr, STRLEN(getErr), 0)
        if logfile.first then
          log ← FAILED_REQUEST_LOG(Headers[Method].values,
           Headers[Filename].values,
           Headers[Protocol].values,
           "404")
          off ← HANDLE_LOGGING_OFFSET(log.length())
          HANDLE_HEADER_LOGGING(log, off, logfiledisc)
        end if
      end if
    else
      inCache ← false
      if (Cache.SIZE() > 0) && caching then
        string DATA
        for (int n ← 0; n != Cache.SIZE(); ++n) do
          fname ← to_string(Headers[Filename].values)
          cachename ← Cache.AT(n).first
            if fname == cachename then
              inCache ← true
              DATA ← Cache.AT(n).second
              break
            end if
        end for loop
        n ← DATA.LENGTH()
        dataArray[n + 1]
        STRCPY(dataArray, DATA.c_str())
        resp[] ← "HTTP/1.1 200 OK\r\nContent-Length: "

        restNum ← TO_STRING(n).LENGTH() + 4
        rest[restNum]
        STRCPY(rest, to_string(n).c_str())
        STRCAT(rest, "\r\n\r\n")
        STRCAT(resp, rest)
        SEND(new_socket, resp, STRLEN(resp), 0)
        WRITE(new_socket, dataArray, STRLEN(dataArray))
      end if
```

```
          if inCache == false then
            total_bytes ← 0
            total_data ← ""
            while (bytes_read ← READ(pfd, buffy, nbytes)) >= 1 do
              total_bytes += bytes_read
              temp(buffy)
              total_data += temp

              l[4]
              SPRINTF(l, "%zu", bytes_read)
              resp[] ← "HTTP/1.1 200 OK\r\nContent-Length: "
              STRCAT(resp, l)
              STRCAT(resp, "\r\n\r\n")
              SEND(new_socket, resp, strlen(resp), 0)
              WRITE(new_socket, buffy, bytes_read)
            end while loop

            if caching then
              pair<string, string> new_file = { Headers[Filename].values, total_data }
              if Cache.SIZE() < 4 then
                Cache.PUSH_BACK(new_file)
              else
                EVICTFILEINCACHE()
                Cache.PUSH_BACK(new_file)
              end if
            end if

            if total_bytes == 0 then
              resp[] ← "HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n"
              SEND(new_socket, resp, STRLEN(resp), 0)
            end if
          end if

          if logfile.first then
             log ← SUCCESSFUL_GET_LOG(Headers[Method].values, Headers[Filename].values,
caching, inCache)
             off ← HANDLE_LOGGING_OFFSET(log.length())
             HANDLE_HEADER_LOGGING(log, off, logfiledisc)
          end if
      end if
      CLOSE(pfd)
   end if

   if validFname && (STRCMP(Headers[Method].values, "PUT") == 0) then
```

```
patty[29]
STRCAT(patty, Headers[Filename].values)
filename 🡐 patty
inDisk 🡐 false
 if ACCESS(fname, F_OK) != -1 then
   inDisk 🡐 true
end if

if (pfd 🡐 open(patty, O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1 then
   Create struct stat s
   if STAT(patty, &s) == 0 then
      mode_t k 🡐 s.st_mode
      if (k & S_IRUSR) == 0 then
         getErr[] 🡐 "HTTP/1.1 403 Forbidden\r\nContent-Length: 0\r\n\r\n"
         SEND(new_socket, getErr, STRLEN(getErr), 0)

         if logfile.first then
            log 🡐 FAILED_REQUEST_LOG(Headers[Method].values,
             Headers[Filename].values,
             Headers[Protocol].values,
             "403")

            PTHREAD_MUTEX_LOCK(&logmutex)
            off 🡐 HANDLE_LOGGING_OFFSET(log.length())
            PTHREAD_MUTEX_UNLOCK(&logmutex)

            // HANDLE_PWRITE
            HANDLE_HEADER_LOGGING(log, off, logfiledisc)
         end if

      else
         putInval[] = "HTTP/1.1 400 Bad Request\r\nContent-Length: 0\r\n\r\n"
         SEND(new_socket, putInval, STRLEN(putInval), 0)
         if logfile.first then
            log 🡐 FAILED_REQUEST_LOG(Headers[Method].values,
             Headers[Filename].values,
             Headers[Protocol].values,
             "400")

            PTHREAD_MUTEX_LOCK(&logmutex)
            off <- HANDLE_LOGGING_OFFSET(log.length())
            PTHREAD_MUTEX_UNLOCK(&logmutex)

            // HANDLE_PWRITE
            HANDLE_HEADER_LOGGING(log, off, logfiledisc)
          end if
        end if
      end if
    else
      Create struct stat file
      if STRCMP("None", Headers[ContentLen].values) == 0 then
        datta[20]
        totalsize ← 0
        resp[] ← "HTTP/1.1 201 Created\r\nContent-Length: 0\r\n\r\n"
        SEND(new_socket, resp, STRLEN(resp), 0)
        While (rdata = READ(new_socket, datta, SIZEOF(datta))) >= 1 do
          totalsize += rdata
          WRITE(pfd, datta, rdata)
        End while

      Else
        rdata ← 0
        l[4]
        SPRINTF(l, "%s", Headers[ContentLen].values)
        if inDisk then
            resp[] ← "HTTP/1.1 201 Created\r\nContent-Length: "
            STRCAT(resp, l)
            STRCAT(resp, "\r\n\r\n")
            SEND(new_socket, resp, STRLEN(resp), 0)
        else
            resp[] ← "HTTP/1.1 200 OK\r\nContent-Length: "
            STRCAT(resp, l)
            STRCAT(resp, "\r\n\r\n")
            SEND(new_socket, resp, STRLEN(resp), 0)
        end if

        len ← ATOI(Headers[ContentLen].values)
        buf ← (char *)MALLOC(len * SIZEOF(char))
        while ATOI(Headers[ContentLen].values) != rdata do
          temp ← READ(new_socket, &datta, 1)
          WRITE(pfd, &datta, temp)
          buf[rdata] = datta
          ++rdata
        End while

        DATA ← “”
        for (i ← 0; i < len; ++i) do
          temp ← buf[i]
```

```
                DATA += temp
            end for

            inCache ← false
            if caching then
                if Cache.SIZE() > 0 then
                    for (n ← 0; n != Cache.SIZE(); ++n) do
                        fname ← to_string(Headers[Filename].values)
                        cachename ← Cache.AT(n).first
                            if fname == cachename then
                                inCache ← true
                                break
                            end if
                    end for loop
                end if

                if inCache then
                    Cache.AT(n).second = DATA
                else
                    pair<string, string> new_file = { Headers[Filename].values, DATA }
                    if Cache.size() < 4 then
                        Cache.push_back(new_file)
                    else
                        evictfileInCache()
                        Cache.push_back(new_file)
                    end if
                end if
            end if

            if logfile.first then
                log ← SUCCESSFUL_PUT_LOG(
                  Headers[Method].values, Headers[Filename].values, Headers[ContentLen].values,
caching, inCache)

                temp ← ATOI(Headers[ContentLen].values) / 20
                tem ← ATOI(Headers[ContentLen].values) % 20
                lines ← tem > 0 ? (temp + 1) : temp
                reservedBytes ← CALCULATETORESERVE(lines, ATOI(Headers[ContentLen].values))
                off ← HANDLE_LOGGING_OFFSET(log.length() + reservedBytes)
                HANDLE_LOG_PUTDATA(log, buf, len, lines, off, logfiledisc)
            end if
            FREE(buf)
        end if
    end if
```

```
        MEMSET(patty, 0, SIZEOF(patty))
        filename ← NULL
        CLOSE(pfd)
    end if

    CLOSE(logfiledisc)
    CLOSE(new_socket)
end handle_connection

procedure httpserver with caching and logging
    port_and_cache ← PARSE_REQUESTS(argc, &(*argv))
    PORT ← port_and_cache.first
    cache ← port_and_cache.second

    addrlen ← SIZEOF(address)
    if (server_fd = SOCKET(AF_INET, SOCK_STREAM, 0)) == 0 then
        PERROR("In socket")
        EXIT(EXIT_FAILURE)
    end if

    address.sin_family ← AF_INET
    if STRCMP(argv[optind], "localhost") == 0 then
        address.sin_addr.s_addr ← inet_addr("127.0.0.1")
    else
        address.sin_addr.s_addr ← inet_addr(argv[optind])
    end if
    address.sin_port ← HTONS(PORT)

    if BIND(server_fd, (struct sockaddr *)&address, SIZEOF(address)) < 0 then
        PERROR("In bind")
        EXIT(EXIT_FAILURE)
    end if

    if LISTEN(server_fd, 10) < 0 then
        PERROR("In listen")
        EXIT(EXIT_FAILURE)
    end if

    while (1) do
        if (new_socket = ACCEPT(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) <
0 then
            PERROR("In accept")
            EXIT(EXIT_FAILURE)
        end if
        HANDLE_CONNECTION(new_socket, cache)
    **end while loop**
    **return** 0
**end procedure**
```