Asgn2 Design Document

Jessica Pan

CruzID: jeypan

CSE130, Fall 2019

1. Goal

The goal of this assignment is to modify the previous HTTP server implemented in Asgn1 into a multithreading server with logging. The server will handle multiple requests simultaneously, each request in its own thread. Logging in this assignment is that the server will write out a record of each request, including both header and data (in hex). There will be the use of synchronization techniques to service multiple requests at once and to ensure the entries in the log aren't inter mixed from multiple threads.

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:
Assuming the threads would run on the server and log data as they complete the request, the order in logging of the requests won't be in order of the requests dequeued

from the queue. The order of the requests logged to the file could be out of order because we don't know which thread would finish before another. However, when they're logged out of order, their data for each request won't be intermixed with another request as each request will use `pwrite()` reserve the exact amount of bytes needed to log their header (and data).

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. 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. In addition, the user will be able to put in the number of threads with the flag '-N' and if the user wants logging done then they'd include the flag '-l' along with the filename. If the number of threads aren't specified then the default number of threads would be 4.

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. But before accepting a client connection, the program will create the number of threads specified prior to running the server with `pthread_create()`. All these threads are stored in a threadpool array and these threads will be asleep. Once all that is done and the client has been accepted, the server will *`pthread_mutex_lock()`* for a socket to enter a thread by enqueueing it into a queue. Then the program will signal with a condition variable using *`pthread_cond_signal()`* letting the program know that there's a thread in the queue. Then the program will dequeue the thread from the queue and have the thread execute what it needs to with a function. After signaling with a condition variable the mutex is unlocked with *pthread_mutex_unlock()*.

Once the thread is dequeued from the queue, the program will 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 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.

In the process of completing these requests, we will also log the request on a file that's specified at the beginning when running the server. If the log file isn't specified then the logging will simply not occur. But if a log file was specified, when we handle each GET/PUT or non-GET/non-PUT method with logging. To log the header (and it's data) contiguously, we will use `pwrite()` which allows us to write to a file starting at a certain position in the file (the offset of however many number of bytes). So, for any request made, there would be an offset that the method would use to log its header (and data). There will be a global offset and this offset will be changed/updated by using `pthread_mutex_lock()` and `pthread_mutex_unlock()`. This way it prevents the threads from having a race condition, where two or more threads could read the same value of offset and write to the same position in the log file. The new offset is calculated by the number of bytes that's need to be written to the file in terms of each request and then that request will take the previous offset value, allowing for no empty spaces in the log file. When logging for GET requests (success/fail), failed PUT requests and non-GET/non-PUT requests, the data logged would just be the header of the request along with the response of the request. As for a successful PUT request, both the header and the data in the file being sent by the client in the PUT request, needs to be logged into the log file. To handle this logging of the header and data, I made a separate function that would first have its local offset and the padding value for the logging. I will read through a buffer (character by character) containing all the data sent by the client in the PUT request. The program will read each character and convert it to its hex value and then place them into a separate buffer and for every 20th character read, the program will log the data into the log file with `pwrite()`, change the local offset and padding value and repeat this process until all the characters in the buffer is read.

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. However, this would not happen with one client-server connection at a time, the program will handle multiple connections with threads which is why we need to use mutex locks and condition variables.

4. Pseudocode

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

```
Function parse_requests (int argc, char* argv[]) do
```

```
    if argc <= 7 then
        while (opt ← GETOPT(argc, argv, "N:l:")) != -1 do
               switch(opt)
                  case 'N':
                     threadpoolSize = atoi(optarg)
                     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_threadpoolSize
    port_and_threadpoolSize.first ← port
    port_and_threadpoolSize.second ← threadpoolSize

    return port_and_threadpoolSize
end parse_requests

Function thread_function(void *arg) do
   while (1) do
      PTHREAD_MUTEX_LOCK(&mutex1)
      if ((pclient ← DEQUEUE()) == NULL) then
         PTHREAD_COND_WAIT(&condition_var, &mutex1)
         pclient ← DEQUEUE()
        end if
        PTHREAD_MUTEX_UNLOCK(&mutex1)
        if (pclient != NULL) then
            HANDLE_CONNECTION(pclient)
        end if
    end while
    return NULL
end thread_function

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) do
    return (method + " " + fname + " length 0\n========\n")
end successful_get_log

Function successful_put_log(string method, string fname, string len) do
    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, SIZEOF(end), tempOffset)
end handle_log_putdata
```

```
Function handle_connection(void* p_new_socket) do
   new_socket ← *((int *)p_new_socket)
   free(p_new_socket)
   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
   valread ← 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")
        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

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")

    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

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")

          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
    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")

                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
    else
        while (bytes_read = READ(pfd, buffy, nbytes)) >= 1 do
            char 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
        if bytes_read == 0 then
            char resp[] = "HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n"
            SEND(new_socket, resp, STRLEN(resp), 0)
        end if
        if logfile.first then
            log ← SUCCESSFUL_GET_LOG(Headers[Method].values, Headers[Filename].values)

            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
    CLOSE(pfd)
end if

if validFname && (STRCMP(Headers[Method].values, "PUT") == 0) then
    patty[29]
```

```
STRCAT(patty, Headers[Filename].values)
filename ← patty

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)

  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)

  len ← ATOI(Headers[ContentLen].values)
  buf ← (char *)MALLOC(len * SIZEOF(char))

  k ← 0
  while ATOI(Headers[ContentLen].values) != rdata do
    temp ← READ(new_socket, &datta, 1)
    WRITE(pfd, &datta, temp)
    buf[k] = datta
    ++rdata
    ++k
  End while

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

    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))

    PTHREAD_MUTEX_LOCK(&logmutex)
    off ← HANDLE_LOGGING_OFFSET(log.length() + reservedBytes)
```

```
                PTHREAD_MUTEX_UNLOCK(&logmutex)

                // HANDLE_PWRITE
                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)
  return NULL
end handle_connection

procedure multithreading httpserver with logging
  port_and_threads ← PARSE_REQUESTS(argc, &(*argv))
  PORT ← port_and_threads.first
  threadpool_size ← port_and_threads.second

  pthread_t threadpool[threadpool_size]

  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

      for (int i = 0; i < threadpool_size; i++) do
         PTHREAD_CREATE(&threadpool[i], NULL, THREAD_FUNCTION, NULL)
      end for loop
      If (new_socket = ACCEPT(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) <
0 then
         PERROR("In accept")
         EXIT(EXIT_FAILURE)
      End if

      pclient ← (int *)MALLOC(SIZEOF(int))
      *pclient ← new_socket
      PTHREAD_MUTEX_LOCK(&mutex1)
      ENQUEUE(pclient)
      PTHREAD_COND_SIGNAL(&condition_var)
      PTHREAD_MUTEX_UNLOCK(&mutex1)
   end while loop
   return 0
end procedure
```