A Project Report On

Multi-Threaded Web Server

Submitted By:
Santosh Kumar Dubey

Under the guidance of:
Prof. Tevfik Kosar

M.S in CS, Semester – I

Department of Computer Science
University at Buffalo

Acknowledgement:

I would like to express my deep sense of gratitude to all those who are associated with my project **"Multi-Threaded Web Server"**, a partial fulfilment of the course curriculum of Operating System.

I would like to acknowledge the support showered with blessings by my dear Professor **Mr. Tevfik Kosar** without whom an opportunity to study and to work on a live project would not have been possible.

I would like to extend my hearty thanks to all TA's of Operating System whose guidance helped us to take right decisions with regards to my project.

OPERATING SYSTEM

CSE – 521 **Date: November -01-2013**
Fall-2013
Santosh Kumar Dubey
Person # 5009 7059

Chapter1
Introduction

The word *multithreading* can be translated as *many threads of control*. While a traditional UNIX process always has contained and still does contain a single thread of control, multithreading (MT) separates a process into many execution threads, each of which runs independently.
Because each thread runs independently, multithreading your code can :

• Improve application responsiveness
• Use multiprocessors more efficiently
• Improve your program structure
• Use fewer system resources
• Improve performance

The goal of the project is to provide a basic web server that should be shielded from all of the details of network connections and the HTTP (Hypertext Transfer Protocol) protocol. Most web browsers and web servers interact using a text-based protocol called HTTP. A web browser opens an Internet connection to a web server and requests some content with HTTP. The web server responds with the requested content and closes the connection. The browser reads the content and displays it on the screen. Each piece of content on the server is associated with a file. If a client requests a specific disk file, then this is referred to as **static content**. If a client requests that an executable file be run and its output returned, then this is **dynamic content**, each piece of content on the server is associated with a file. Each file has a unique name known as a URL (Universal Resource Locator).
For example, the URL www.cse.buffalo.edu:80/index.html identifies an HTML file called “index.html” on Internet host “www.cs.buffalo.edu” that is managed by a web server listening on port 80. Each file has a unique name known as a URL (Universal Resource Locator).
For example, the URL www.cse.buffalo.edu:80/index.html identifies an HTML file called “index.html” on Internet host “www.cse.buffalo.edu” that is managed by a web server listening on port 80.
An HTTP request (from the web browser to the server) consists of a request line, followed by zero or more request headers, and finally an empty text line. A request line has the form: method uri version. The method is usually GET (but may be other things, such as POST, OPTIONS, HEAD or PUT).
Finally, the version indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0). An HTTP response (from the server to the browser) is similar; it consists of a response line, zero or more response headers, an empty text line, and finally the interesting part, the response body. A response line has the form version status message. The status is a three-digit positive integer that indicates the state of the request; some common states are

- 200 for OK ,
- 403 for Forbidden ,
- 404 for Not found.

Two important lines in the header are Content-Type, which tells the client the MIME type of the content in the response body (e.g., html or gif) and Content-Length, which indicates its size in bytes.

1.1 Objectives:

The main objectives behind this project is to :

• Creating and synchronizing cooperating processes and threads in Linux.
• Setting up client-server inter-process communication using Internet sockets.
• Applying different scheduling and overload handling strategies to improve system performance.
• Using a benchmarking tool to measure the system performance.
• Learning how a basic web server is structured.
• Learning the basic elements of the World Wide Web and its protocols.

1.2 Requirement:

Implement a multithreaded web server “myhttpd” in C/C++ on a UNIX-based platform.

myhttpd [−d] [−h] [−l file] [−p port] [−r dir] [−t time] [−n threadnum] [−s sched]

−d : Enter debugging mode. That is, do not daemonize, only accept one connection at a time and enable logging to stdout. Without this option, the web server should run as a daemon process in the background.

−h : Print a usage summary with all options and exit.

−l *file* : Log all requests to the given file. See LOGGING for details.

−p *port* : Listen on the given port. If not provided, **myhttpd** will listen on port 8080.

−r *dir* : Set the root directory for the http server to *dir.*

−t *time* : Set the queuing time to *time* seconds. The default should be 60 seconds.

−n *threadnum*: Set number of threads waiting ready in the execution thread pool to *threadnum.* The default should be 4 execution threads.

−s *sched* : Set the scheduling policy. It can be either **FCFS** or **SJF**. The default will be **FCFS**.

Chapter 2
System Analysis

2.0 INFORMATION GATHERING:

I gathered information and needs of the system by talking to college professor and by referring many internet resources to accomplish this project. To get more information about this system I conducted brainstorming sessions with various students from university who were also working on this project.
After the details analysis I came up with different approach and solutions. Keeping those views in minds below is some important problem statements which I tried to resolved in my project:

- Deadlock
- Race Condition
- Handling Log of errors
- Overload Handling
- Context Switch between Threads
- Handling Producer and consumer synchronization mechanism

Firstly I gathered all the necessary and mandatory information about the entire above mentioned topic and then I planned accordingly to implement the code. Once above problem have solved I moved on next like :

- To handle the types of request (GET or POST) have been made.
- To handle the types of contents.
- To maintain the system in consistent state.
- To handle the situation like if after making request from client side and client get disconnected than to handling this kind of situation condition.
- Maintaining the Threads
- Maintaining the List and queue.
- Taking the request from browser and to give proper response.

While analyzing the system I came across many situations where I required having enough information to handle the situation. Firstly I used and maintain the request queue through Array and ArrayList but it was not feasible to us in the system because on its inconsistent state and security concerns I removed it and used Linked List.
In this Phase firstly I cleared all my concern doubts regarding the implementation and design of the system properly.

Chapter 3
System Planning

Project Progression/Planning Chart :

Project Name : Multi-Threaded Web Server

Total Duration of the project (Days) : 36 Days (September/29/2013 to November/04/2013)

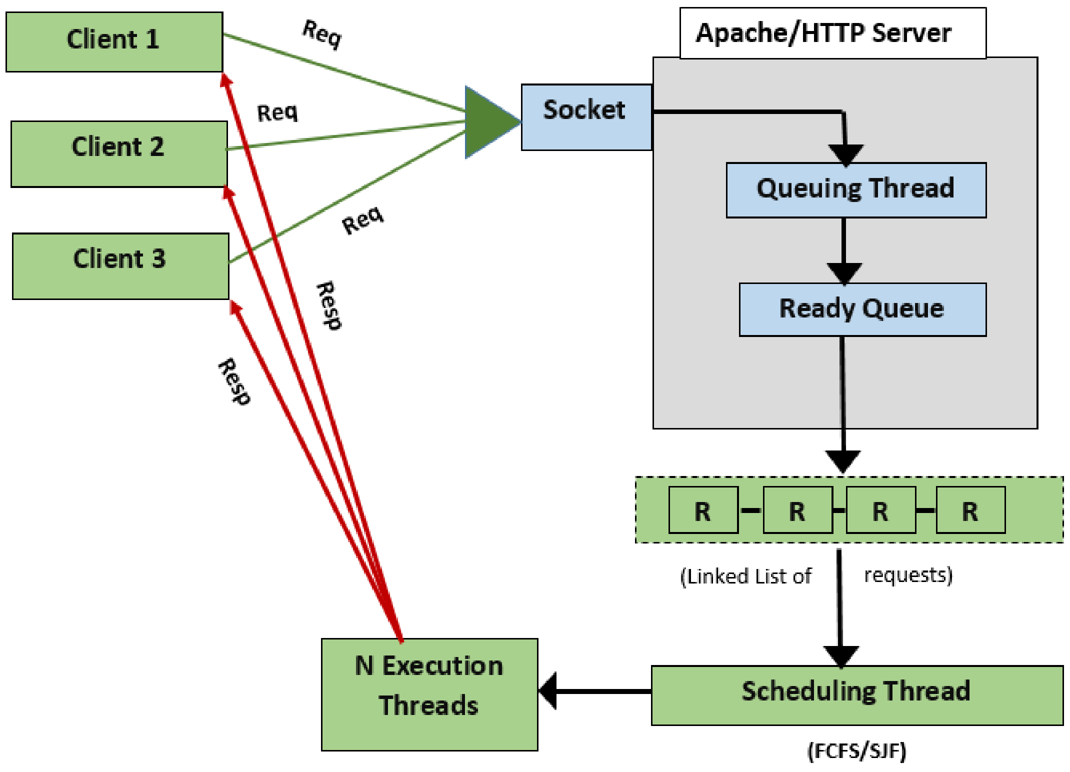
Task	Duration in Days	Fall - 2013						
		Sept 29 – Oct 3	Oct 3 – Oct 7	Oct 7 – Oct 10	Oct- 10 – Oct 15	Oct 15 - Oct 25	Oct 25 – Oct 27	Oct 10 - Nov 01
Requirement Specification	5	🟦						
Planning	5		🟦					
Information Gathering	4			🟦				
Design	6				🟦			
Implementation (coding)	11					🟦		
Testing	3						🟦	
Documentation	22				🟦	🟦	🟦	🟦

Note:

1. Duration in days also Includes end date in calculation (1 day is added)
2. Documentation of the project was based on progression of the project. Thus it took 22 days.

Chapter 4
System Design

4.0 Block Diagram :



Above block diagram gives an overview of design of multithreaded webserver. Client 1, 2 & 3 are any PC's or Browser windows which communicate with the server to get the http data.

Below is the description of the Individual Module / Block,

- **Clients:** Clients are either individual machines in the LAN of the server or the browser of the same machine as server which can be used for demonstrating the implementation. Multiple clients can send the requests to the server, which can serve the client as per command line scheduling policy.

- **Server:** Server is the c programmed implementation for listening requests from different clients as per port mentioned in the command line or a default custom port preprogramed into the server code. Server implements the basic socket programming, where a server blocks on listen to accept the connect request from clients. Once the request from specific client comes, server pushes the concerned request into ready queue.

- **Scheduling Thread:** This is a Linux POSIX thread, which takes the request from client as per mentioned scheduling policy from command line and pushes the request to the ready queue.

- **N execution Threads :** The server program while starting with argument of N threads from command line, creates the N POSIX threads, which can be used to serve the requests from the multiple clients, as mentioned in above block diagram.

Chapter 5
System Implementation

<u>Implementation :</u>

The project is implemented in C language using standard Network Socket programing, Apache / HTTP web protocol, Linux POSIX threading.

Logical implementation of handling command line arguments as per requirement are as below, following code shows the implementation of daemon related command line argument handling, i.e. If "-d" argument is given with "myhttd" (./myhttpd -d)

```
static struct option long_options[] = {
            {"daemon", no_argument, 0, 'd'},
            {0, 0, 0, 0}
};

while(1) {
            c = getopt_long(argc, argv, "dhl:p:r:t:n:s:", long_options, &option_index);
            if (c == -1)
                        break;

            switch(c) {
            case 0 :
                        if (long_options[option_index].flag != 0)
                        break;
            case 'd':
                        printf("Starting myhttpd in debug mode\n");
                        do_daemonize = false;
                        break;
            }
}
```

If the argument "-d" is passed from the command line, the starting of the program is done in interactive mode i.e. Debug messages / errors will be printed to stdout. If we do not pass "-d" argument, the program will be started as a "daemon" i.e. Background process, (do_daemonize = false , from above)

logical implementation of the daemon is as below, where we create a new process using "fork" system call, which act as creating a daemon process.

```
if (do_daemonize) {
            pid = fork();
            if (pid < 0) {
                        exit(EXIT_FAILURE);
            } else
                        exit(EXIT_SUCCESS);
            close(STDIN_FILENO); close(STDOUT_FILENO); close(STDERR_FILENO);
}
```

by default, if we do not provide "-d", the program gets started as a daemon process.

5.1 Basic Server Implementation :

The server is based on standard UNIX network socket programming.

```
sock_id = socket(AF_INET, SOCK_STREAM, 0);
if (sock_id == -1)
            return  error;

setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

memset((void *) &saddr, 0, sizeof(saddr));
saddr.sin_port = htons(portnum);
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = INADDR_ANY;

if (bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr)) != 0) {
            return error;
}

if (listen(sock_id, backlog) != 0)
            return error;
else
            return sock_id;
```

Above is the logical implementation of starting a server, binding it to the either **default 8080 port** or port number given from the command line while starting the program & listening on the socket for the connection request from the clients.

5.2 Queuing Thread :

The queuing thread is a POSIX thread, and the basic purpose of this thread is to accept the request from remote client and insert the client into ready queue.

The details of every new client from which the http request comes, will be stored in structure "struct clientDetails", details of this structure are as below,

```
struct clientDetails {
      int fd;
      struct sockaddr_in addr;
      int length;
      char * filename;
      char * ipaddress;
      char * message;
};
```

where,

fd – is a client file descriptor as rerurned from "accept" socket call.
addr – received client socket address
length – size of the file requested from client as on server filesystem
filename – name of file requested from client as on server filesystem
ipaddress – human readable ip address of client
messsage – request as received from client

Above structure gives the details of the single client, which is being accepted by the server listening on the particular socket, for receiving number of simultaneous requests from different clients, we have to maintain the queue.

<u>Details about Queuing thread :</u>

Queuing thread is a POSIX thread, and will be crated from main program while starting of the program,

```
pthread_t queuing_tid;
pthread_create(&queuing_tid,NULL,&queuing_thread,&sock);
```

where, "pthread_create" is a standard POSIX API, which is used to create a thread, "queuing_tid" is a thread identification which can be used to track the thread, the main program needs to be waited to finish the execution of the queuing thread using standard "pthread_join" API as , pthread_join(queuing_tid, &ret) where, "queuing_tid" is thread id as described above, "ret" stores return value from thread to check valid / successful completion

"queuing_thread" is a thread handler function which is getting called, when the thread is created, the details about this function are as below,

```
void *queuing_thread(void *arg) {
        int sock = *(int *)arg;
}
```

Here, the argument passed to queuing thread, is a socket which is created from main program and which is used for accepting the client request.

```
while (1) {
            clientfd = accept(sock,(struct sockaddr*)&cliaddr, &clientlength);
}
```

Above implementation shows "accept" blocking call, which is blocked till the request comes from a new client, once the request comes, "accept" receives this client and informs about the unique client file descriptor for further operation of handling this client.

Once the servers accepts the client, we need to read the request and process as per the requirement, reading can be done by standard "recv" function as, recv(clientfd,BUF,BUFFER_SIZE,0); where the received message is stored in the buffer "BUF" of size "BUFFER_SIZE", once the message has been received we have to extract the client information from the message and prepare a client structure, which needs to be added to the ready queue for the processing.

```
struct clientDetails *client_ptr;
client_ptr = (struct clientDetails *)malloc(sizeof(struct clientDetails));
client_ptr->ipaddress = inet_ntoa(cliaddr.sin_addr);
client_ptr->dispatched_time = time(0);
client_ptr->addr = cliaddr;
client_ptr->fd = clientfd;
client_ptr->length = size;
client_ptr->filename = file;
client_ptr->message = BUF;
```

The details of above implementation as described above in description of "truct clientDetails". Once the client details has been extracted, we need to add this client to the ready queue, details of the ready queue are as described below,

5.3 Details about Ready Queue:

```
struct ready_queue {
	struct clientDetails *client;
	struct ready_queue *next;
};
```

This is a single linked list implementation of the queue, which grows as and when the requests comes from the clients.

Once as new client request comes, we have to allocate a memory to the client, and add this new client to the end of the queue for processing by scheduling / execution thread,

```
new = (struct ready_queue *)malloc(sizeof(struct ready_queue));
new->client = client;
new->next = NULL;

if(head == NULL) {
		head = new;
		rear = head;  /* This is start node */
} else {
		rear->next = new;
		rear = new;
}
```

5.4 Details about scheduling thread :

Scheduling thread is a POSIX thread, and will be created from main program while starting of the program,

```
pthread_t scheduling_tid;
pthread_create(&scheduling_tid,NULL,&schduling_thread,NULL);
```

where, “pthread_create” is a standard POSIX API, which is used to create a thread, “scheduling_tid” is a thread identification which can be used to track the thread, the main program needs to be waited to finish the completiong of the scheduling thread using standard “pthread_join” API as , pthread_join(queuing_tid, &ret) where, “queuing_tid” is thread id as described above, “ret” stores return value from thread to check valid / successful completion

“scheduling_thread” is a thread handler function which is getting called, when the thread is created, the details about this function are as below,

```
void *scheduling_thread(void *arg) {
		while(1) {
				pthread_mutex_lock(&queue_mutex);
				pthread_cond_wait(&req_came_to_queuing_thread_condition, &queue_mutex);
				if(strcmp(sched_policy, "FCFS") == 0)
						g_client = get_client_fcfs();
				else if(strcmp(sched_policy, "SJF") == 0)
						g_client = get_client_sjf();
		pthread_mutex_unlock(&queue_mutex);
		pthread_cond_signal(&execution_thread_condition);
	}
}
```

Above is the logical implementation of the scheduling thread, the scheduling thread is responsible for removing of the client from the ready queuing which is maintained by the “queuing thread”,

5.5 Protecting critical section & synchronization :

The ready queue will be shared between queuing thread & scheduling thread, so the two threads needs to in synchronization to avoid the corrupting data & getting the results as expected, for that standard mutex have been used, which prevents the unexpected access of shared ready queue, for that we are using following API's

```
 pthread_mutex_lock(&queue_mutex);

... implementation of accessing shared ready queue ...

 pthread_mutex_unlock(&queue_mutex);
```

Actual implementation logic is as per above section of code from "queue_thread" function.

5.6 Synchronization :

The synchronization between different threads needs to be done using standard pthread library functions, pthread_cond_wait & pthread_cond_signal, the below section of code, give logical implementation of synchronization between queuing thread, scheduling thread and execution thread,

```
/* The below function waits for the request to code to ready queue */
pthread_cond_wait(&req_came_to_queuing_thread_condition, &queue_mutex);
            .... handle scheduling ....
            .... report to execution thread that we have a client needs to be handled ...
pthread_cond_signal(&execution_thread_condition);
```

The pthread library function, "pthread_cond_signal" is used to report to execution thread that we have got one client which needs to be handled.

5.7 Scheduling Policies :

Since we will not have any control over which thread is actually scheduled at any given time by the OS. So scheduling is use to determine which HTTP request should be handled by each of the waiting worker threads in web server. In order to improve the throughput, a web server may implement different scheduling policies. A scheduling policy that the server would use is determined by a command line argument when the web server is started.
Following are scheduling policies which are used in this project :

1) FCFS (First Come First Served) :

First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion. This is default scheduling policy.

2) SJF (Shortest Job First) :

Shortest Job First (SJF) scheduling is a priority and Non Preventive. Non Preveentive mean here is when the allotted time a processor then the processor cannot be taken the other, until the process is completed in the execution. This assumes scheduling time to complete the process known in advance. The mechanism is to schedule the process with the shortest time to completion first, thus providing high efficiency and low turnaround time. In terms of time spent in the current program (job) began to enter into the system until the process is completed the system, requiring a short time. Shortest Job First (SJF) scheduling algorithm can be said to be optimal with an average waiting time is minimal.

The scheduling policy can be decided from command line argument while starting program “myhttpd”, if not provided from command line, default scheduling policy will be “First Come First Served [FCFS] “, the logical implementation of handling command line argument for scheduling policies is as below,

```
static struct option long_options[] = {
        {"sched", required_argument, 0,  's'}
}

c = getopt_long(argc, argv, "s:", long_options, &option_index);

switch(c) {
        case 's':
            sched_policy = optarg;
            break;
}
```

while starting “myhttpd” application, if we provide “./myhttpd -s FCFS” for “First Come First Served” scheduling policy or “./myhttpd -s SJF” for “Shorted Job First” scheduling policy, the local variable “ sched_policy” stores the type of policy and is passed to scheduling thread, the necessary scheduling of clients from ready queue will be done as below & based on “sched_policy” as provided,

```
if(strcmp(sched_policy, "FCFS") == 0) {
        g_client = get_client_fcfs();
 } else if(strcmp(sched_policy, "SJF") == 0) {
        g_client = get_client_sjf();
  }
```

In above implementation, functions “get_client_fcfs” will have implementation necessary for removing client from ready queue as per “First Come First Served (FCFS)” scheduling policy whereas, “get_client_sjf” will have implementation necessary for removing client from ready queue as per “Shortest Job First (SJF)” scheduling policy.

5.8 n * execution threads :

The number of execution threads can be decided from command line argument while starting program “myhttpd”, if not provided from command line, default number of threads will be 4 as per the requirement. The logical implementation of handling command line argument for thread numbers is as below,

```
static struct option long_options[] = {
            {"threadnum", required_argument, 0,  'n'}
}

c = getopt_long(argc, argv, "n:", long_options, &option_index);

switch(c) {
            case 's':
                  threadnum = atoi(optarg);
                  break;
}
```

while starting “myhttpd” application, if we provide “./myhttpd -n threadnum” where “threadnum” is “n” numbers of threads as per the requirement.

The standard pthread library “pthread_create” will be used to create number of execution threads, whereas “pthread_join” function will be used to wait for the threads to complete. For practical convenience we are using max numbers of threads which can be created to 1024, which can be changed while programing as per the requirement,

```
# define  MAX_NUM_THREADS 1024
pthread_t execution_tid[MAX_NUM_THREADS]

for (i=0; i < threadnum; i++) {
            pthread_create(&execution_tid[i],NULL,&execution_thread,NULL);
}

for (i=0; i < threadnum; i++) {
            if(pthread_join(execution_tid[i], &ret))
                        printf("unable to wait for execution thread completion\n ");
}
```

The logical implementation of “execution thread” will be as described below,

```
void *execution_thread(void *arg) {
            struct clientDetails *cli;

            while (1) {
                        pthread_cond_wait(&execution_thread_condition, &queue_mutex);

                        cli = g_client->client;
            if(cli != NULL) {
                                    process_rq(cli);
                                    free(g_client); /*we have served this client, now free its memory*/
                        }
            }
}
```

“pthread_cond_wait” library function is used for synchronization with the “scheduling thread” using the, mutex “queue_mutex”, the details of this synchronization are as described above respective section.

The function “process_rq” does the actual processing of the client message / request and is responsible for sending response to the respective client. The “process_rq” does the HTTP header processing, checks the type of the file, and sends the requested information, if available to the respective socket. Logical implementation of this function is as below,

```
void process_rq(struct clientDetails *cli) {
        char *rq = cli->message;

        if (sscanf(rq, "%s %s", cmd, arg + strlen(run_dir)) != 2)
                return;

        if (strcmp(cmd, "GET") != 0) {
                printf("can not process HTTP Request \n");
                return error;
        } else {
        send_file(arg, fd);
        }
}
```

<u>**5.9 Logging :**</u>

Logging mechanism is used for debugging. As per request from command line argument, logging can be either printed to the console [-d argument] or can be redirected to the request file [-l file]

Logging can be enable compile time with a macro ENABLE_LOGS or if command line argument [-l file] is used while starting server, logging will be enabled by default and logs are saved to file in filesystem.

```
#define ENABLE_LOGS 1

#ifdef ENABLE_LOGS

        fprintf(fp, “%a %t %t %r %s %b\n”, remote_ip, time_recv, time_assign, request, status, resp_size );

#endif
```

Above, fp can be set to file pointer of stdout (if -d is given while starting myhttpd) or to file pointer of “file” (if “-l file” is used while starting myhttpd)

```
 /* Logging */
log_fp = fopen(log_file, "w+");
if (!log_fp) {
        log_fp = stdout;
        printf("unable to open log file, continuing with stdout\n");
}

fprintf(log_fp, "started daemon successfully\n");
```

The above code shows the logical implementation of appending the logs to the file, if valid [-l file] command line argument is given from the command line, while starting the program.

Chapter 6
System Testing & Compilation

Compilation of Project:

There is a makefile written to get myhttpd compile and create a executable, hence to compile source code just type "make" command in terminal and press enter,
Some steps to follow :

<u>Step 1:</u> Creating executable file of "myhttp" by using make command.

```
$ make

user@user-desktop:~/multithreaded-http$ make

gcc -g -Wall myhttpd.c -o myhttpd -lpthread
```

<u>Step 2 :</u>

```
user@user-desktop:~/ ls -al

-rwxrwxr-x 1 user user 37632 Oct 17 13:02 myhttpd
```

This shows, make command created an executable "myhttpd" of size "37632" bytes, executing and starting this program has been explained in README.

<u>Step-3:</u> Staring Server

```
$ ./myhttpd
```

This command will start server and wait for request to be send from browser or Terminal as client.
In this all the following command can be passed as an argument to test different functionality of the program and output.

<u>Step 4 :</u> Sending request from browser to the server by typing following command .

```
http//localhost:8080/index.html
```

This command will display the requested file on browser and requested file will be send by the server, if file is not found then it will show appropriate error to the user on the browser.

<u>Step 5 :</u> Sending GET/HEAD request from Terminal to the server by typing following command .

a) GET /index.html HTTP/1.0

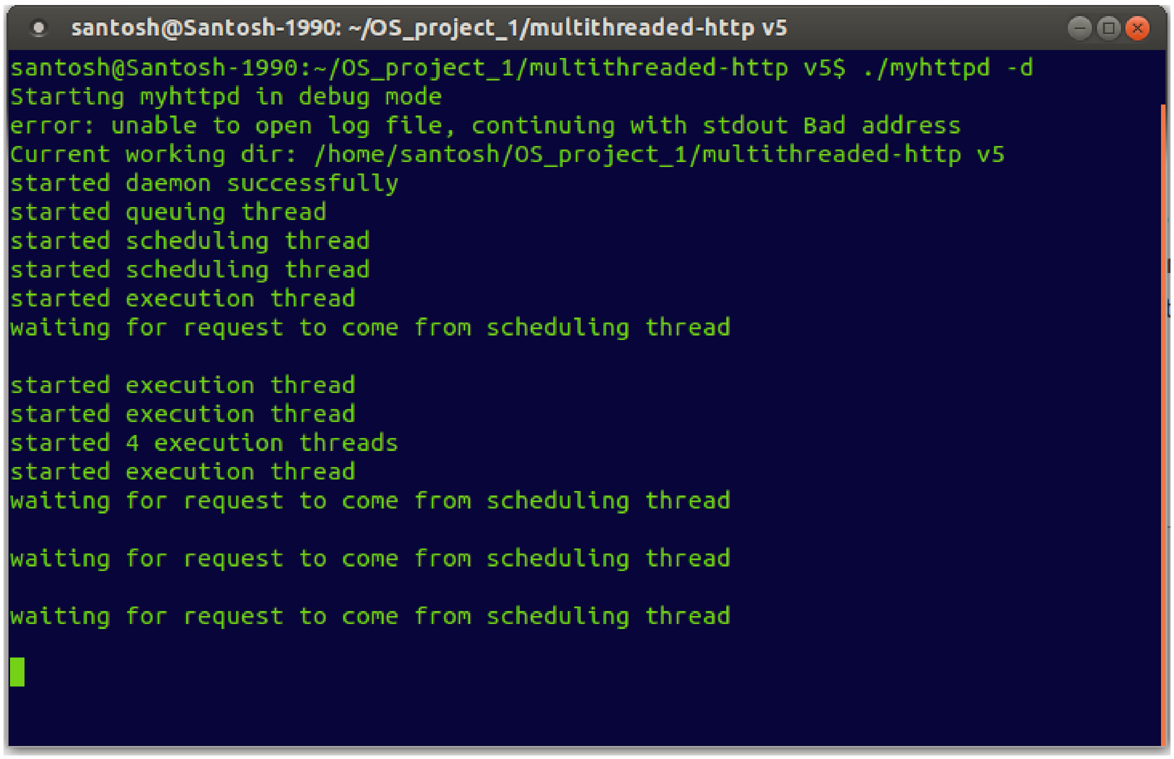
This command will retrieve the requested file from server and display it on the screen.

b) HEAD /index.html HTTP/1.0

This command will display the metadata of the requested file

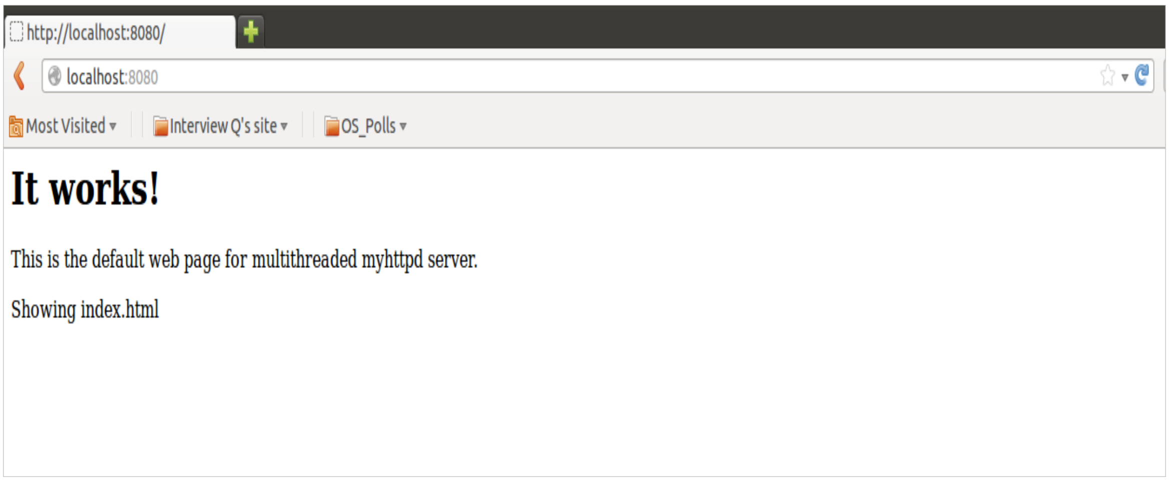
Screenshots :

1. Starting Server



```
santosh@Santosh-1990:~/OS_project_1/multithreaded-http v5$ ./myhttpd -d
Starting myhttpd in debug mode
error: unable to open log file, continuing with stdout Bad address
Current working dir: /home/santosh/OS_project_1/multithreaded-http v5
started daemon successfully
started queuing thread
started scheduling thread
started scheduling thread
started execution thread
waiting for request to come from scheduling thread

started execution thread
started execution thread
started 4 execution threads
started execution thread
waiting for request to come from scheduling thread

waiting for request to come from scheduling thread

waiting for request to come from scheduling thread
```

2. Starting Browser to request index.html file



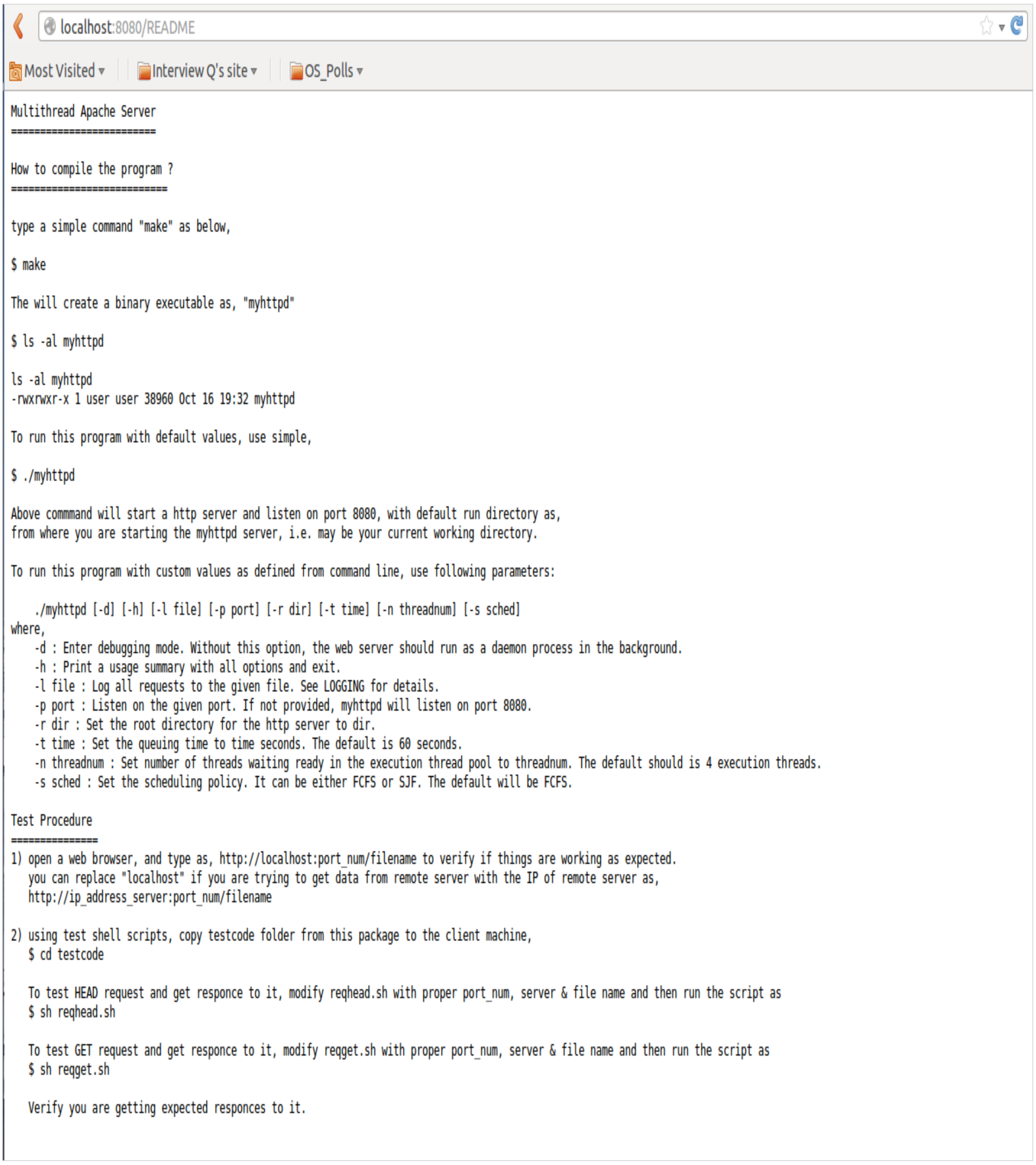
It works!

This is the default web page for multithreaded myhttpd server.

Showing index.html

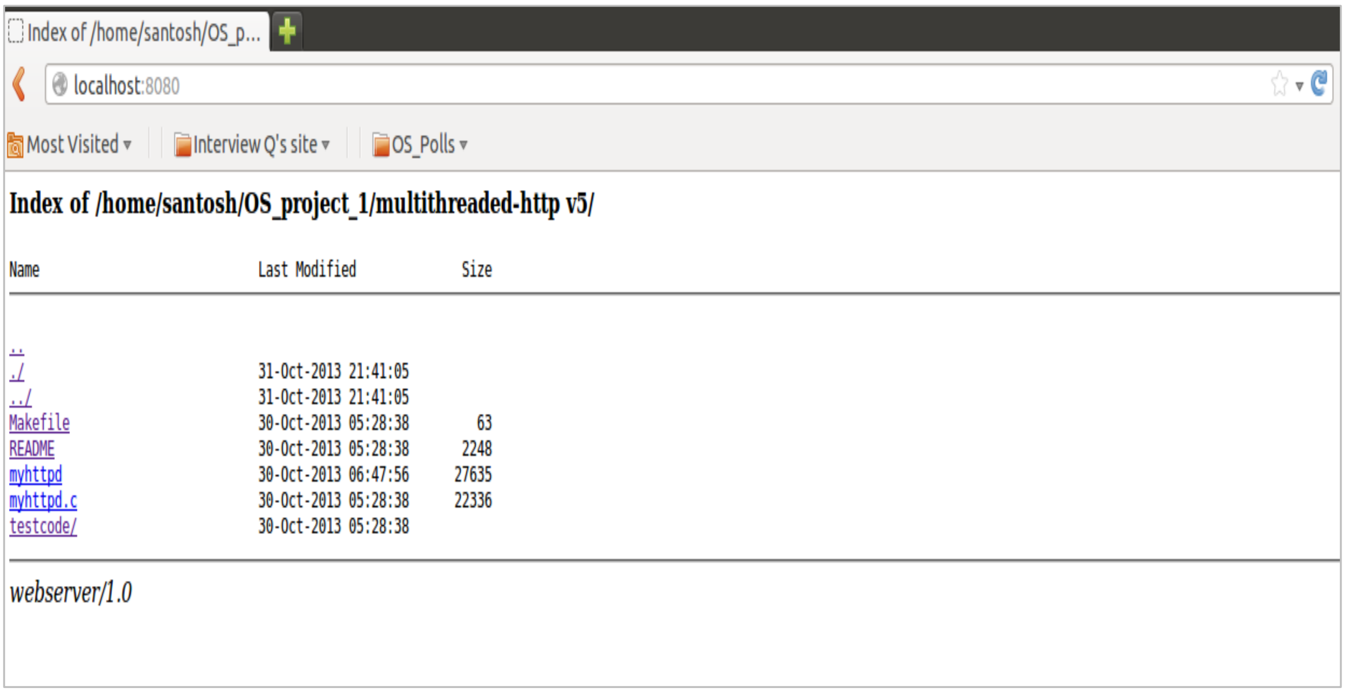
3. Requesting README file from browser

Open a browser, and typed localhost:8080/README to get README file over http.



```
Multithread Apache Server
=========================

How to compile the program ?
============================

type a simple command "make" as below,

$ make

The will create a binary executable as, "myhttpd"

$ ls -al myhttpd

ls -al myhttpd
-rwxrwxr-x 1 user user 38960 Oct 16 19:32 myhttpd

To run this program with default values, use simple,

$ ./myhttpd

Above command will start a http server and listen on port 8080, with default run directory as,
from where you are starting the myhttpd server, i.e. may be your current working directory.

To run this program with custom values as defined from command line, use following parameters:

    ./myhttpd [-d] [-h] [-l file] [-p port] [-r dir] [-t time] [-n threadnum] [-s sched]
where,
    -d : Enter debugging mode. Without this option, the web server should run as a daemon process in the background.
    -h : Print a usage summary with all options and exit.
    -l file : Log all requests to the given file. See LOGGING for details.
    -p port : Listen on the given port. If not provided, myhttpd will listen on port 8080.
    -r dir : Set the root directory for the http server to dir.
    -t time : Set the queuing time to time seconds. The default is 60 seconds.
    -n threadnum : Set number of threads waiting ready in the execution thread pool to threadnum. The default should is 4 execution threads.
    -s sched : Set the scheduling policy. It can be either FCFS or SJF. The default will be FCFS.

Test Procedure
===============
1) open a web browser, and type as, http://localhost:port_num/filename to verify if things are working as expected.
   you can replace "localhost" if you are trying to get data from remote server with the IP of remote server as,
   http://ip_address_server:port_num/filename

2) using test shell scripts, copy testcode folder from this package to the client machine,
   $ cd testcode

   To test HEAD request and get responce to it, modify reqhead.sh with proper port_num, server & file name and then run the script as
   $ sh reqhead.sh

   To test GET request and get responce to it, modify reqget.sh with proper port_num, server & file name and then run the script as
   $ sh reqget.sh

   Verify you are getting expected responces to it.
```

4. Showing Directory if index.html is not present.

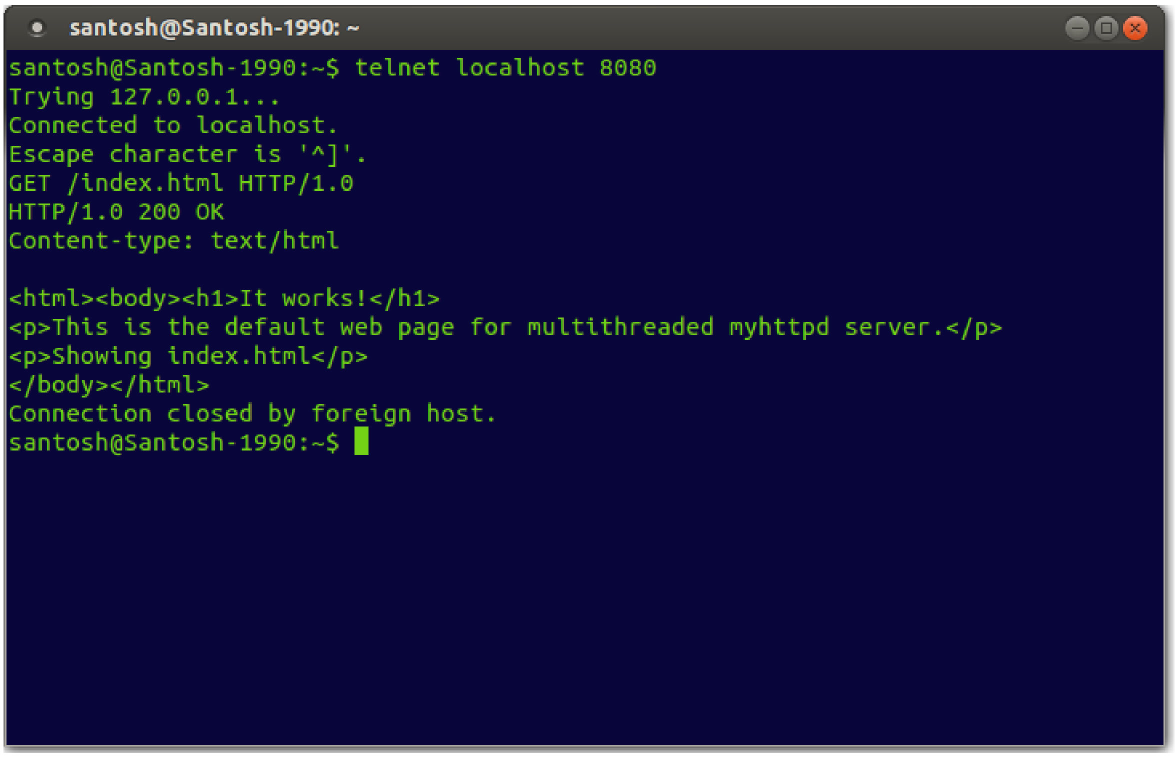


Index of /home/santosh/OS_project_1/multithreaded-http v5/

```
Name                Last Modified          Size

..
./                  31-Oct-2013 21:41:05
../                 31-Oct-2013 21:41:05
Makefile            30-Oct-2013 05:28:38     63
README              30-Oct-2013 05:28:38   2248
myhttpd             30-Oct-2013 06:47:56  27635
myhttpd.c           30-Oct-2013 05:28:38  22336
testcode/           30-Oct-2013 05:28:38
```

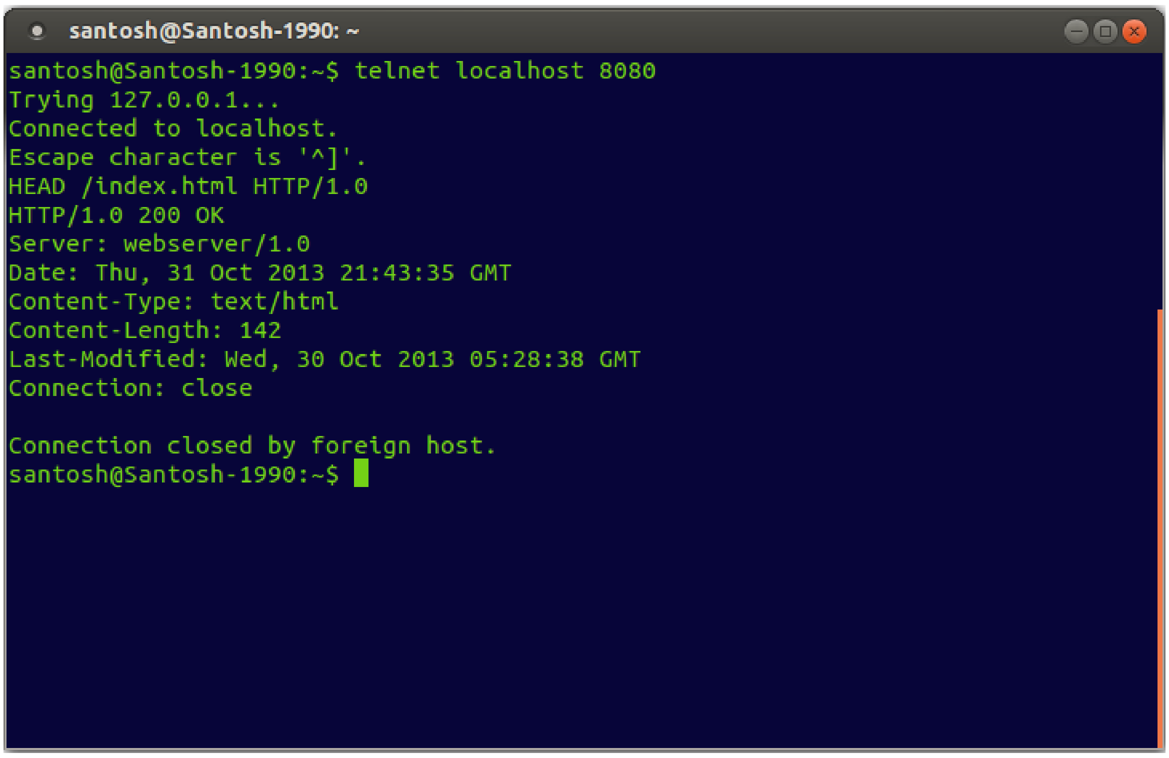
webserver/1.0

5. Requesting through Terminal using GET method



```
santosh@Santosh-1990:~$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /index.html HTTP/1.0
HTTP/1.0 200 OK
Content-type: text/html

<html><body><h1>It works!</h1>
<p>This is the default web page for multithreaded myhttpd server.</p>
<p>Showing index.html</p>
</body></html>
Connection closed by foreign host.
santosh@Santosh-1990:~$ 
```

6. Requesting through Terminal using HEAD method


santosh@Santosh-1990: ~

```
santosh@Santosh-1990:~$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HEAD /index.html HTTP/1.0
HTTP/1.0 200 OK
Server: webserver/1.0
Date: Thu, 31 Oct 2013 21:43:35 GMT
Content-Type: text/html
Content-Length: 142
Last-Modified: Wed, 30 Oct 2013 05:28:38 GMT
Connection: close

Connection closed by foreign host.
santosh@Santosh-1990:~$ 
```

Chapter 7

Q & A Session

1) About Deadlock

Answer:

Deadlocks happen if a mutex is acquired and never released, but code is taking care of releasing a mutex every time, hence there might not be a situation of deadlock coming.

2) Why MAX number of threads has been defined in program?

Answer:

the theoretical creation of unlimited threads is not possible in practical, since as we create more threads, whole system can get slowed by using as much as RAM, which is not a desired condition, hence for practical implementation there has to be some limitation of MAX THREADS created, currently it is set to 1024, but is required it can be change as much as possible as required by changing #define in source before compilation.

3) Handling Log error of arguments.

Answers:

All failures are appended in to log file if given [-l log_file] option, or to standard out if no [-l log_file] mentioned.

4) How you are finding out which request has been made among GET and HEAD? Also how your requested file and links has been parsed and shown the output on web browser?

Answers:

You can use command as, "curl -i -X HEAD http://localhost:port-num/file-name" to use "HEAD" whereas "curl -i -X GET http://localhost:port_num/file_name"
Since browsers mostly doesn't support HEAD, you have to use command line "curl" for requesting head information.

5) If I closed server (by closing terminal cmd) and after this if I type localhost:8080/<filename> then also why I am getting all requested file if my server is closed?

Answer:
Since you are starting myhttpd as daemon, the process gets detached from terminal, and becomes as background process, hence even if you close terminal, program keeps on running, so to close it completely do "ps -ax", know the process id of "./myhttpd" and kill it manually, "sudo kill -9 process_id_of_myhttpd" but if you start it as "./myhttpd -d" then if you close terminal, servers also gets stopped automatically

6) When updating port number by typing command ./myhttpd -p 9090 then it should not work on predefined port 8080 it should only work on new updated port number.

Answers :

"ps -ax" command can be used to check whether the server is running or not
if you have already running a daemon myhttpd with 8080, and start another in new terminal with 8181, you will get proper result on both 8080 & 8181 port, so in this case make sure to kill the daemon using "ps -ax; then note process id of myhttpd & kill it using "sudo kill -9 process_id" this should keep only one myhttpd server running.

Chapter 7

Source Code

```
/*
 * Multithread myhttpd server
*/

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#include <getopt.h>
#include <pthread.h>

#include <dirent.h>

#define true 1
#define false 0

FILE *log_fp = NULL;
char *run_dir;

pthread_cond_t execution_thread_condition;
pthread_cond_t req_came_to_queuing_thread_condition;
pthread_mutex_t exec_mutex;
pthread_mutex_t queue_mutex;

#define MAX_NUM_THREADS 1024

#define LOGGING 1

#ifdef LOGGING
#define LOG(str) fprintf(log_fp, str);
#endif

#define BACKLOG 10

int run_dir_is_home_dir = 0;

/* structure for storeing clinet information */
struct clientDetails
{
	int fd;
	struct sockaddr_in addr;
```

```
        int length;
        char * filename;
        char * ipaddress;
        char * message;
            char * firstline;
        int status;
        time_t dispatched_time;
            time_t executed_time;
};

/* Linked list istructure of ready queue which contains accepeted clients */
struct ready_queue
{
        struct clientDetails *client;
        struct ready_queue *next;
};

struct ready_queue *new, *temp,*p,*head=NULL,*rear=NULL;

char BUF[1024];
int wait_time = 60; /* default waiting time of 60 sec */

/* default scheduling to FCFS */
char* sched_policy = "FCFS";

/* basic setup of http server socket */
int make_server_socket_q(int portnum, int backlog)
{
            struct sockaddr_in saddr;
            int sock_id;
            socklen_t opt = 1;

            sock_id = socket(AF_INET, SOCK_STREAM, 0);
            if (sock_id == -1) {
                        perror("call to socket");
                        return -1;
            }

            setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

            memset((void *) &saddr, 0, sizeof(saddr));
            saddr.sin_port = htons(portnum);
            saddr.sin_family = AF_INET;
            saddr.sin_addr.s_addr = INADDR_ANY;

            if (bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr)) != 0) {
                        perror("call to bind");
                        return -1;
            }

            if (listen(sock_id, backlog) != 0) {
                        perror("call to listen");
                        return -1;
            } else {
```

```
                        return sock_id;
            }
}

int make_server_socket(int portnum) {
            return make_server_socket_q(portnum, BACKLOG);
}

void do_404(const char *item, int fd) {
            FILE *fp = fdopen(fd, "w");

            fprintf(fp, "HTTP/1.0 404 Not Found\r\n");
            fprintf(fp, "Content-type: text/plain\r\n");
            fprintf(fp, "\r\n");
            fprintf(fp, "The requested URL %s is not found on this server.\r\n", item);
            fclose(fp);
}

/* Cant handle this request, so send appropriate error response */
void canot_do(int fd) {
            FILE *fp = fdopen(fd, "w");

            fprintf(fp, "HTTP/1.0 501 Not Implemented\r\n");
            fprintf(fp, "Content-type: text/plain\r\n");
            fprintf(fp, "\r\n");
            fprintf(fp, "That command is not yet implemented\r\n");
            fclose(fp);
}

/* Extract the extension of the file requested from the client */
char *get_mime_type(const char *name) {
  char *ext = strrchr(name, '.');
  if (!ext) return NULL;
  if (strcmp(ext, ".html") == 0 || strcmp(ext, ".htm") == 0) return "text/html";
  if (strcmp(ext, ".jpg") == 0 || strcmp(ext, ".jpeg") == 0) return "image/jpeg";
  if (strcmp(ext, ".gif") == 0) return "image/gif";
  if (strcmp(ext, ".png") == 0) return "image/png";
  if (strcmp(ext, ".css") == 0) return "text/css";
  if (strcmp(ext, ".au") == 0) return "audio/basic";
  if (strcmp(ext, ".wav") == 0) return "audio/wav";
  if (strcmp(ext, ".avi") == 0) return "video/x-msvideo";
  if (strcmp(ext, ".mpeg") == 0 || strcmp(ext, ".mpg") == 0) return "video/mpeg";
  if (strcmp(ext, ".mp3") == 0) return "audio/mpeg";
  return NULL;
}

/* Send the response to HEAD request from client */
void do_head(const char*f, FILE *fp, int status, char *title, int length)
{
            time_t now;
            char timebuf[128];
            time_t date;
            struct stat statbuf;
```

```
#define SERVER "webserver/1.0"
#define PROTOCOL "HTTP/1.0"
#define RFC1123FMT "%a, %d %b %Y %H:%M:%S GMT"

        if (stat(f, &statbuf) >= 0)
                date = statbuf.st_mtime;

        fprintf(fp, "%s %d %s \r\n", PROTOCOL, status, title);
        fprintf(fp, "Server: %s\r\n", SERVER);
        now = time(NULL);
        strftime(timebuf, sizeof(timebuf), RFC1123FMT, gmtime(&now));
        fprintf(fp, "Date: %s\r\n", timebuf);

        fprintf(fp, "Content-Type: %s\r\n", get_mime_type(f));
        if (length >= 0) fprintf(fp, "Content-Length: %d\r\n", length);

        if (date != -1) {
                strftime(timebuf, sizeof(timebuf), RFC1123FMT, gmtime(&date));
                fprintf(fp, "Last-Modified: %s\r\n", timebuf);
        }
        fprintf(fp, "Connection: close\r\n");
        fprintf(fp, "\r\n");
}


void header(FILE *fp, const char *content_type)
{
        fprintf(fp, "HTTP/1.0 200 OK\r\n");
        if (content_type) {
                fprintf(fp, "Content-type: %s\r\n", content_type);
        }
}

/* Check if file requested from client is present on the server or not */
int not_exist(const char *f)
{
        struct stat *info = (struct stat *)malloc(sizeof(struct stat));

        if (stat(f, info) == -1) {
                free(info);
                return 1;
        } else {
                free(info);
                return 0;
        }
}

/* Check if file requested from client is a directory */
int isadir(const char *f)
{
        struct stat *st = (struct stat *)malloc(sizeof(struct stat));;

        if (stat(f, st) != 0) {
                free(st);
                return 0;
        } else {
                if(S_ISDIR(st->st_mode)) {
                        free(st);
                        return 1;
                } else {
                        free(st);
                        return 0;
                }
        }
}

/* view & send the file's content to remote client */
void do_cat(const char *f, int fd)
{
        char *content = get_mime_type(f);
        FILE *fpsock = NULL;
        FILE *fpfile = NULL;
        int c;

        fprintf(log_fp, "sending file : %s\n", f);

        fpsock = fdopen(fd, "w");
        fpfile = fopen(f, "r");
        if (fpsock != NULL && fpfile != NULL) {
                header(fpsock, content);
                fprintf(fpsock, "\r\n");

                while ((c = getc(fpfile)) != EOF) {
                        putc(c, fpsock);
                }

                fclose(fpfile);
                fclose(fpsock);
        }
}

/* If file requested from client is a direcory
  Sort the direcories and send the directory structure to the client
  which can be visible to the client in browser */
int do_ls(const char *path, int fd, int length)
{
        FILE *fp;
        int len = strlen(path);
        struct stat statbuf;
        char pathbuf[4096];

     struct dirent *de;
        struct dirent **namelist;
        int i=0, n=0;


        snprintf(pathbuf, sizeof(pathbuf), "%sindex.html", path);
        if (stat(pathbuf, &statbuf) != 0) {
                printf("file or dir not present: %s\n", pathbuf);
//              return 0;
        }

        if (S_ISREG(statbuf.st_mode)) {
                do_cat(pathbuf, fd);
                fprintf(log_fp, "is a regular file\n");
                return 0;
        }

        fp = fdopen(fd, "w");
        do_head(path, fp, 200, "OK", length);
    fprintf(fp, "<HTML><HEAD><TITLE>Index of %s</TITLE></HEAD>\r\n<BODY>",
path);
    fprintf(fp, "<H4>Index of %s</H4>\r\n<PRE>\n", path);
    fprintf(fp, "Name                          Last Modified             Size\r\n");
    fprintf(fp, "<HR>\r\n");
    if (len > 1) fprintf(fp, "<A HREF=\"..\">..</A>\r\n");

        /* used for sorting of directory listing */
        n = scandir(path, &namelist, NULL, alphasort);

         if (n < 0)
                perror("scandir");
        else {
                for (i=0; i<n; i++ ) {
                        de = namelist[i];
                        char timebuf[32];
                        struct tm *tm;

                        strcpy(pathbuf, path);
                        strcat(pathbuf, de->d_name);

                        stat(pathbuf, &statbuf);
                        tm = gmtime(&statbuf.st_mtime);
                        strftime(timebuf, sizeof(timebuf), "%d-%b-%Y %H:%M:%S", tm);

                        fprintf(fp, "<A HREF=\"%s%s\">", de->d_name,
S_ISDIR(statbuf.st_mode) ? "/" : "");
                        fprintf(fp, "%s%s", de->d_name, S_ISDIR(statbuf.st_mode) ?
"/</A>" : "</A> ");
                        if (strlen(de->d_name) < 32) fprintf(fp, "%*s", (int)(32 -
strlen(de->d_name)), "");

                        if (S_ISDIR(statbuf.st_mode))
                                fprintf(fp, "%s\r\n", timebuf);
                        else
                                fprintf(fp, "%s %10d\r\n", timebuf, (int)statbuf.st_size);

                        free(namelist[i]);
                }
                free(namelist);
        }
```

```
      fprintf(fp,
"</PRE>\r\n<HR>\r\n<ADDRESS>%s</ADDRESS>\r\n</BODY></HTML>\r\n",
SERVER);
        fclose(fp);
        return 0;
}

/*
 * skip over all request info until a CRNL is seen
 */
void read_til_crnl(FILE *fp)
{
        char buf[BUFSIZ] = { 0 };
        while (fgets(buf, BUFSIZ, fp) != NULL && strcmp(buf, "\r\n") != 0);
}

/*
 * Check who is the current logged in user, from which the server is started
 * This is used to send the response if ~ is used in the browser
*/
char *get_home_dir (void) {
        char *buf = (char *)malloc(1024*sizeof(char));
        if(buf == NULL) {
                fprintf(log_fp, "error: allocating memory for
buffer %s\n",strerror(errno));
                exit(errno);
        }
        char *temp = (char *)malloc(256*sizeof(char));
        if(temp == NULL) {
                fprintf(log_fp, "error: allocating memory for temp %s\n",strerror(errno));
                free(buf);
                exit(errno);
        }

        strcpy(buf, "/home/");
        cuserid(temp);
        strcat(buf, temp);
        strcat(buf, "/myhttpd");

        free(temp);
        return buf;
}

/* Handle the request HEAD/GET from the client and
  send the appropriate response
 */
void process_rq(struct clientDetails *cli)
{
        char cmd[BUFSIZ] = {0};
        char arg[BUFSIZ] = {0};
        char *rq = cli->message;
        int fd = cli->fd;

        cli->executed_time = time(0);
	strcpy(arg, run_dir);

	if (sscanf(rq, "%s %s", cmd, arg + strlen(run_dir)) != 2)
		return;

	if (run_dir_is_home_dir) {
		strcpy(arg, cli->filename);
		run_dir_is_home_dir = 0;
	}

	fprintf(log_fp, "cmd == %s, arg=%s\n", cmd, arg);

	if (strcmp(cmd, "GET") != 0) {
		if (strcmp(cmd, "HEAD") == 0) {
			FILE *fp = fdopen(fd, "w");
			fprintf(log_fp, "do HEAD\n");
			do_head(arg, fp, 200, "OK", cli->length);
			fclose(fp);
		} else {
			fprintf(log_fp, "can not do\n");
			canot_do(fd);
		}
	} else if (not_exist(arg)) {
		fprintf(log_fp, "not exist\n");
		do_404(arg, fd);
	} else if (isadir(arg)) {
		fprintf(log_fp, "is a dir\n");
		do_ls(arg, fd, cli->length);
	} else {
		fprintf(log_fp, "do cat\n");
		do_cat(arg, fd);
	}
}

/* Print help */
void help(void) {
	printf("\n\t============================ Run this server as
==================================\n\n");
	printf("\tmyhttpd [-d] [-h] [-l file] [-p port] [-r dir] [-t time] [-n threadnum] [-s
sched]");
	printf("\n\n");
	printf("\t[-d] Enter debugging mode, do not daemonize\n");
	printf("\t[-h] Print usage summary\n");
	printf("\t[-l file] Log all requests to given file\n");
	printf("\t[-p port] Listen on the given port\n");
	printf("\t[-r dir] Set the root dir\n");
	printf("\t[-t time] Set the queuing time to time seconds\n");
	printf("\t[-n threadnum] Set number of threads waiting in execution thread\n");
	printf("\t[-s sched] Set the scheduling policy, FCFS / SJF\n\n");
}

/* Used for debugging purpose to check how many cients are in ready queue */
void display_queue()
```

```
{
            fprintf(log_fp, "displaying queue \n");

       if(head == NULL)
             fprintf(log_fp, "empty queue");
       else {
              temp = head;
              while(temp != NULL)
            {
                   fprintf(log_fp, "\n acceptfd is %d, file name is %s, ip addr is %s\n",
                                       temp->client->fd,temp->client->filename,
temp->client->ipaddress);
                              fprintf(log_fp, "message: %s\n", temp->client->message);
                   temp = temp->next;
            }
       }
}

/* Insert client into the linked list implementation of the ready queue */
void insert_to_ready_queue(struct clientDetails * client) {

            new = (struct ready_queue *)malloc(sizeof(struct ready_queue));
            new->client = client;
            new->next = NULL;

            if(head == NULL) {
                        head = new;
                        rear = head; /* This is start node */
            }
       else {
              rear->next = new;
              rear = new;
            }
}

/* Formatted Logging */
void log_http_request(struct clientDetails *client) {

       //fprintf(log_fp, "log:");
       char logbuf[4096] = {0};
       char firstline[4096];
            int i=0;
            struct clientDetails *temp = client;

            char logstatus[10];
            char loglength[10];

            fprintf(log_fp, "log request: ");
       strcpy(logbuf, client->ipaddress);
       strcat(logbuf," -  ");

       strcat(logbuf, ctime(&client->dispatched_time));

       strcat(logbuf, ctime(&client->executed_time));
```

```
        while (*temp->firstline != '\n') {
                firstline[i++] = *temp->firstline;
                temp->firstline++;
      }

        firstline[i] = '\0';

      strcat(logbuf, firstline);

        sprintf(logstatus,"%d",client->status);
        sprintf(loglength,"%d",client->length);

        strcat(logbuf," ");
        strcat(logbuf,logstatus);
        strcat(logbuf," ");
        strcat(logbuf,loglength);

        fprintf(log_fp, "%s", logbuf);
}

char dir_Name[1024];
#define BUFFER_SIZE 1024

/* Implementation of queuing thread  */
void *queuing_thread(void *arg) {
        int sock = *(int *)arg;

      struct clientDetails *client_ptr;
      int n,size;
        struct stat *st = NULL;
      struct sockaddr_in cliaddr;
      char  *file = (char *)malloc(4096),reldir[4096];
      int clientfd;
        socklen_t clientlength = sizeof(struct sockaddr_in);
        char *home_dir;
        char *charposition = NULL;
        char *temp = (char *)malloc(4096);

        while (1) {

                st = malloc(sizeof(struct stat));
                if(st == NULL) {
                        fprintf(log_fp, "error: allocating memory for
client_ptr %s\n",strerror(errno));
                        exit(errno);
                }

                /* Accept remote client */
         clientfd = accept(sock,(struct sockaddr*)&cliaddr, &clientlength);
         if(clientfd < 0) {
                        fprintf(log_fp, "error: accepting client %s\n",strerror(errno));
                        free(st);
              exit(2);
```

```
            }

            if(clientfd > 0) {
                        n = recv(clientfd,BUF,BUFFER_SIZE,0);//max_size
                if(n == 1) {
                              fprintf(log_fp, "error: receive %s\n",strerror(errno));
                        }

                if(n == 0) {
                    //close(clientfd);
                    fprintf(log_fp, "directory requested\n");
                }

                sscanf(BUF,"%*s %s",file);

                        if ( (charposition = strchr(file, '~')) != NULL) {
                              strcpy(temp, (charposition + 1));
                              home_dir = get_home_dir();
                              strcpy(file, home_dir);
                              strcat(file, "/"); /* attached received filename at the end to
skip ~ */
                              strcat(file, temp); /* attached received filename at the end to
skip ~ */
//                            free(home_dir);
                              run_dir_is_home_dir = 1;
                        }

                        strcpy(reldir, run_dir);
                        strcat(reldir, file); /* make file as relative path dir */

                stat(reldir, st);
                size = st->st_size;
                        free(st); /* Free memory used for getting info from stat */
                client_ptr = (struct clientDetails *)malloc(sizeof(struct clientDetails));
                        if(client_ptr == NULL) {
                              fprintf(log_fp, "error: allocating memory for
client_ptr %s\n",strerror(errno));
                              exit(errno);
                        }
                client_ptr->ipaddress = inet_ntoa(cliaddr.sin_addr);
                client_ptr->dispatched_time = time(0);
                client_ptr->addr = cliaddr;
                client_ptr->fd = clientfd;
                client_ptr->length = size;
                client_ptr->filename = file;
                        client_ptr->message = BUF;
                        client_ptr->firstline = BUF;
                        client_ptr->status = 1;
                        fprintf(log_fp, "server: got connection from %s, file=%s\n",
client_ptr->ipaddress, client_ptr->filename);

                        insert_to_ready_queue(client_ptr);
                        pthread_cond_signal(&req_came_to_queuing_thread_condition); /*
Signal to Scheduling Thread that we have one more client */
```

```
                        }
            }

            fprintf(log_fp, "completed queuing thread\n");
}

/* Remove clinet from ready queue as per FCFS scheduling policy */
struct ready_queue *get_client_fcfs(void) {
            struct ready_queue *temp = NULL;;

            if(head == NULL)
              fprintf(log_fp, "empty queue\n");
       else {
                        temp = head;
              head = head->next;
       }

            return(temp);
}

int identify_shortest_job(void) {
            int shortestjob_fd = 0;
            int min, b;

            temp = head;

            if (temp == NULL) {
                        fprintf(log_fp, "SJF Queue is Empty\n");
            } else if(temp->next == NULL) {
                        /* This has only one client */
                        shortestjob_fd = temp->client->fd;
            } else {
                        min = temp->client->length;
                        while(temp->next != NULL) {
                                    b = temp->next->client->length;
                                    if(min <= b) {
                                                shortestjob_fd = temp->client->fd;
                                    } else if (min > b) {
                                                min = temp->next->client->length;
                                                shortestjob_fd = temp->next->client->fd;
                                    }
                                    temp = temp->next;
                        }
            }
            fprintf(log_fp, "returning fd %d\n", shortestjob_fd);
            return shortestjob_fd;
}

/* Remove clinet from ready queue as per SJF scheduling policy */
struct ready_queue* get_client_sjf(void) {
            int shortestjob_fd;
            struct ready_queue* previous = NULL;

            shortestjob_fd = identify_shortest_job();
```

```
        if (!shortestjob_fd) {
          fprintf(log_fp, "SJF Queue is Empty\n");
      } else {
          temp = head;
          while(temp != NULL) {
                        if(temp->client->fd == shortestjob_fd) {

                                if(temp == head)
                                        head = temp->next; /* update head */
                                else {
                                        previous->next = temp->next;
                                }
                                return temp;
                        } else {
                                previous = temp;
                                temp = temp->next; /* Go to next client in the list */
                        }
                }
        }
        return NULL;
}

struct ready_queue *g_client = NULL; /* Global Pointer Shared between scheduling &
Execution Thread */

/* Implementation of execution thread  */
void *execution_thread(void *arg) {
        struct clientDetails *cli;
        fprintf(log_fp, "started execution thread\n");

        while (1) {
                pthread_mutex_lock(&queue_mutex);

                fprintf(log_fp, "waiting for request to come from scheduling thread\n\n");

                pthread_cond_wait(&execution_thread_condition, &queue_mutex);
                fprintf(log_fp, "signal received into execution thread, start processing\n");

                if (g_client != NULL)
                        cli = g_client->client;

                /* Remove client from ready queue, as per FCFS scheduling policy */
                if(cli != NULL) {
                        process_rq(cli);
                        log_http_request(cli);
                        //free(cli); /*we have served this client, now free its memory*/
                        free(g_client); /*we have served this client, now free its memory*/
                }

              pthread_mutex_unlock(&queue_mutex);
        }
        fprintf(log_fp, "completed execution thread\n");
}
```

```
/* Implementation of Scheduling thread  */
void *scheduling_thread(void *arg) {

	fprintf(log_fp, "started scheduling thread\n");

	while(1) {
		pthread_mutex_lock(&queue_mutex);
		pthread_cond_wait(&req_came_to_queuing_thread_condition,
&queue_mutex);

		if(strcmp(sched_policy, "FCFS") == 0) {
			g_client = get_client_fcfs();
		} else if(strcmp(sched_policy, "SJF") == 0) {
			g_client = get_client_sjf();
		}
		pthread_mutex_unlock(&queue_mutex);
		printf("request came to queuing thread, signal execution thread to execute
it\n");
		pthread_cond_signal(&execution_thread_condition);
	}
	fprintf(log_fp, "completed scheduling thread\n");
}

int main(int argc, char *argv[])
{
	int sock;
	int c, threadnum = 4, i=0;
	static int option_index;
	char *log_file;
	int portnum = 8080, do_daemonize = true;
	int use_default_dir = 1;

	pthread_t queuing_tid, execution_tid[MAX_NUM_THREADS], scheduling_tid;
	void* ret = NULL;

	run_dir = (char *)malloc(1024); /* Default Run Dir is pwd */
	if(run_dir == NULL) {
		fprintf(stderr, "error: allocating memory for
run_dir: %s\n",strerror(errno));
		exit(errno);
	}

	/* Structure for handling command line arguments as used while starting main
program */
	static struct option long_options[] = {
		{"daemon", no_argument, 0, 'd'},
		{"help", no_argument, 0, 'h'},
		{"log_file", required_argument, 0, 'l'},
		{"port", required_argument, 0, 'p'},
		{"rundir", required_argument, 0,  'r'},
		{"time", required_argument, 0,  't'},
		{"threadnum", required_argument, 0,  'n'},
		{"sched", required_argument, 0,  's'},
		{0, 0, 0, 0}
	};

	while(1) {
		c = getopt_long(argc, argv, "dhl:p:r:t:n:s:", long_options, &option_index);
		if (c == -1)
			break;

		switch(c) {
		case 0 :
			if (long_options[option_index].flag != 0)
			break;
		case 'd':
			printf("Starting myhttpd in debug mode\n");
			do_daemonize = false;
			break;
		case 'h':
			help();
			exit(0);
		case 'l':
			log_file  = optarg;
			break;
		case 'p':
			portnum = atoi(optarg);
			break;
		case 'r':
			use_default_dir = 0;
			run_dir = optarg;
			break;
		case 't':
			break;
		case 'n':
			threadnum = atoi(optarg);
			break;
		case 's':
			sched_policy = optarg;
			break;
		}
	}

	/* Logging */
	log_fp = fopen(log_file, "w+");
	if (!log_fp) {
		log_fp = stdout;
		fprintf(log_fp, "error: unable to open log file, continuing with
stdout %s\n",strerror(errno));
	}

	/* use current directory as default directory for accessing */
	if (use_default_dir) {
		if (getcwd(run_dir, 1024) != NULL){
			fprintf(log_fp, "Current working dir: %s\n", run_dir);
		}
	}
	/* Start this application as daemon */
	if (do_daemonize) {
		fprintf(log_fp, "starting myhttpd daemon\n");
		daemon(1, 1);
	}

	fprintf(log_fp, "started daemon successfully\n");

	sock = make_server_socket(portnum);
	if (sock == -1) {
		fprintf(log_fp, "error: creating socket %s\n",strerror(errno));
		exit(2);
	}

	/* Initialize mutex and condition variable objects */
	pthread_mutex_init(&exec_mutex, NULL);
	pthread_mutex_init(&queue_mutex, NULL);
	pthread_cond_init (&execution_thread_condition, NULL);
	pthread_cond_init (&req_came_to_queuing_thread_condition, NULL);

	/* Queuing Thread Implementation */
	pthread_create(&queuing_tid,NULL,&queuing_thread,&sock);
	fprintf(log_fp, "started queuing thread\n");
	/* End of Queuing Thread Implementation */

	/* Scheduling Thread Implementation */
	pthread_create(&scheduling_tid,NULL,&scheduling_thread, NULL);
	fprintf(log_fp, "started scheduling thread\n");

	/* Execution Thread Implementation, start n thread as per command line
argument */
	for (i=0; i < threadnum; i++) {
		pthread_create(&execution_tid[i],NULL,&execution_thread,NULL);
	}
	fprintf(log_fp, "started %d execution threads\n", i);

	/* wait for thread to complete */
	if(pthread_join(queuing_tid, &ret))
		fprintf(log_fp, "error: unable to wait for queuing thread
completion :%s\n", strerror(errno));

	for (i=0; i < threadnum; i++) {
		if(pthread_join(execution_tid[i], &ret))
			fprintf(log_fp, "error: unable to wait for execution thread
completion :%s\n", strerror(errno));
	}

	if(pthread_join(scheduling_tid, &ret))
		fprintf(log_fp, "error: unable to wait for scheduling thread
completion :%s\n", strerror(errno));

	/* Distroy The Mutexes */
	pthread_mutex_destroy(&exec_mutex);
	pthread_mutex_destroy(&queue_mutex);
```

```
        pthread_cond_destroy(&execution_thread_condition);
        pthread_cond_destroy(&req_came_to_queuing_thread_condition);

        fprintf(log_fp, "exiting from main\n");
        return 0;
}
```

Web references :

1. **RFC for HTTP**
   http://tools.ietf.org/html/rfc1945

2. **Guide to network Programming using Network Sockets -**
   http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

3. **Linux POSIX Threads**
   http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

4. **HTTP Tutorials**
   http://www.tutorialspoint.com/http/http_methods.htm

5. **Multi-threaded multi-process web server**
   http://httpd.apache.org/docs/2.2/mod/worker.html

6. **PHOSIX Threads Programming**
   https://computing.llnl.gov/tutorials/pthreads/

7. **Pthreads**

   http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread.h.html

8. **Pthreads API**
   http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Fapis%2Frzah4mst.htm

9. **Multi-Threaded Programming With POSIX Threads**
   http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html#thread_mutex_destroy

10. **POSIX Threads Tutorial by Mark Hays**
    http://www.laptev.org/doc/pthreads.html