# PROJECT 1

**HTTP WEBSERVER with multithreading and scheduling**

Name: Dhruv Seth

COURSE: CSE521 - Operating Systems

Date: 20th Oct 2011

## Overview

It is a multithreaded web server “myhttpd” in C on a UNIX-based platform. Myhttpd is a simple web server. It binds to a given port on the given address and waits for incoming HTTP/1.0 requests.

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

The webserver is based on client-server socket programming. Where the server is listens to the request on a port and serves it to the client via a socket connection.

## HTTP Webserver design:

1) Parsing switches from the command line.

2) Starting the server in daemon or debug mode.

3) Parse the request.

4) Execute the request (depending upon the mode)

### Parsing the switches from the command line:

The function "readParameters()" is used to parse the command line arguments given by the shell. It sets the parameters for the http webserver. Parsing the modes (daemon or debug), logging mechanism, default webserver directory, queuing time, defining the listening port, number of execution threads to be used and the scheduling algorithm.

### Starting the server:

The modes defined in the program are mode=1 (for daemon mode) and mode=0 (for debug mode). The daemon mode is the default mode of the web server unless change to debug mode by using -d switch. If the debug mode is required the number of threads are automatically changed to one and the scheduling algorithm is set to FCFS (First come first serve). Also, the queuing time is reduced to 1 second as the request has to be executed immediately.

### Parse the request:

The request which is picked up by the execution thread is parsed using the “cmdType” field in the buffer. If the “cmdType” field is “GET” the thread is instructed to do a CAT of the file or list the directory’s files in an alphabetical order. If the “cmdType” is “HEAD” the request to just to send the HTTP Header with the HTTP Status ie. 200 if the request is OK or 404 if the request is BAD.

### Execute the request:

The execution thread opens the FILE pointer of the file descriptor in the request buffer. The field “clientfd” is the file descriptor and is used to open a new FILE pointer. Using this file pointer, the request is sent to the client.

### The two modes:

* DAEMON MODE
* DEBUG MODE

In the DAEMON MODE, the http webserver program is forked whenever the function to start the server "main()" is called. Thereby, creating a different process which takes care of the entire http server and it runs in background. This mode has various options such as to increase the number of execution threads, change the scheduling policy, enable/disable logging mechanism, set the listening port and change the time of queuing the requests. Apart from this the switch -r <dir> can be used to change the default directory of the webserver.

In DEBUG MODE, the http webserver program is run as a foreground process, with default scheduling (FCFS) and one execution thread. It waits for the client request and executes it immediately. It has options such as to enable/disable logging mechanism and set the listening port, change the default directory of the webserver.

## HTTP server has number of parts:

* Two global queues managed by semaphores.
* One thread to listen to the requests.
* One thread to schedule the requests.
* 4 (or user defined) threads to execute the requests.

The queues used are basically structures arrays which have elements with datatype:

1. int filesize;
2. char cmdType[4];
3. char requesttype[10];
4. char path[SIZE];
5. int clientfd;
6. time\_t arrivaltime;
7. time\_t responsetime;
8. char clientip[250];

filesize: stores the size of the file requested by the client/browser

cmdType: is used to store the HTTP request/command viz. GET or HEAD

requesttype: is used to store the status of the HTTP request viz LS, CAT, BAD. LS indicated it has to display the directory, CAT indicates that it has to display the file and BAD indicates it an invalid request.

path: is the path of the file/directory requested.

clientfd: it is the file descriptor using which the request is being processed and sent back.

arrivaltime: it is the arrival time of the reqeuest. Used for logging purposes.

responsetime: it is used for logging purposes and stores the time when the request was executed and sent.

clientip: it is used to store the IP address of the client. Used for logging purposes only.

The two global queues are used. One is to keep the requests, sort them and move them to the next queue for execution by the execution threads. The listening thread keeps filling the first (named rdybuffer[]) queue and scheduling thread sorts (bubble sort used) the requests and inserts them into the execution (named exbuffer[]) queue. These queues are controlled by two mutex locks, one for listening thread and scheduling thread to access the rdybuffer[] queue and the other mutex lock is shared between scheduling thread and the execution threads to allow them to control the exbuffer[] queue.

## Synchronization:

To synchronize the two queues Semaphores used are:

* p\_s\_emptybuffers and p\_s\_fullbuffers -->used between listening thread and scheduling thread
* s\_ex\_emptybuffers and s\_ex\_emptybuffers -->used between scheduling thread and execution threads.
* p\_s\_mutex is used as a binary semaphore to control listener and scheduler threads over ready queue (rdybuffer[]).
* s\_ex\_mutex is used as a binary semaphore to control scheduler and execution threads over execution queue(exbuffer[]).

## Scheduling:

The scheduling is done on if and only if the policy is SJF, else the thread just returns the latest request and pushes it to the execution queue for the execution threads to process it. If the SJF is chosen the scheduling thread sorts (uses bubble sort) the rdybuffer[] queue before returning the shortest (or lowest file size) job. Which is then pushed to the execution buffer and fed to the execution threads.

## Multithreading:

The threads are engaged when the mode of the web server is set. If the mode is debugging mode these threads are started: listening thread (named producer), scheduler thread and one execution thread. If the mode is daemon, the threads are initiated with number of execution threads set by the user. The function "serverfunc" contains the thread initialization code. Each thread has thread functions:

* Listening thread has function: "producerfunc" which is invoked when it starts.
* Scheduler thread has function: "schedulerfunc".
* Execution thread has funciton: "consumerfunc" which is used by execution threads to serve the requests.

## Logging mechanism:

The webserver enables logging to a log file if and only if the mode is daemon mode. If debug mode is running logging lines will be displayed on the terminal. By default, myhttpd does not do any logging. If explicitly enabled via the -l flag.

README

=======

Author: Dhruv Seth

Dated: 15th Oct 2011

INSTALLATION

===========

Run the command "make" on the directory you wish to install the program. The output will be a myhttpd binary file.

Root directory

============

The httpd web server starts with the default directory "/myhttpd" If one wants to change the default directory use the -r <dir> switch. For example,

./myhttpd -r /myhttpd/dir1

The webserver will set it's root path to this directory, if it exists.

MAX EXECUTION THREADS

=====================

The maximum number of threads is set to 20. One can change the #define MAXTHREADS value in the source.c

404 (page not found) page

=======================

The page has been created such as it would display the path requested by the client and the webserver's root directory.

# SOURCE CODE

## Source.c

#include "stdio.h"

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/stat.h>

#include <pthread.h>

#include <semaphore.h>

#include <dirent.h>

#include <fcntl.h>

#define PORTNUMBER 8080

#define SER\_BUFF\_SIZE 1000

#define LOGFILE "/myhttpd/logfile"

#define DIRC "/myhttpd"

#define SIZE 200/\* Directory to fetch files \*/

#define ARGSIZE 1000

#define READYQUEUESIZE 10

#define MAXTHREADS 20

#define DIR\_SIZE 500

/\*dhruv seth\*/

/\* and global variables \*/

static int port = PORTNUMBER; //default port number

static char RootDirectory[SIZE] = DIRC; //default server directory

static int qtime = 60; //default wait time

static int eThread = 4; //default number of execution threads

static int Schd = 1; //FCFS = 1 & SJF = 2 (default = 1) & 0 means off

static int mode = 0; //regular mode = 0 (default ie with threads) & debugging mode = 1 & helpmode =2

static int logmode =0; // no logging = 0(default) & yes logging = 1 & logging =2 on stdout

static int sfd; //Global server's File Descriptor

static char LogFile[SIZE] = LOGFILE; //default log file

static int rdyq\_currentsize = 0;

struct readyqueueBUFF

{

int markforexecution; //0 for no; 1 for yes -- not being used

int filesize;

char cmdType[4];

char requesttype[10]; //LS or CAT or BAD

char path[SIZE];

int clientfd;

time\_t arrivaltime;

time\_t responsetime;

char clientip[250];

};

// information to maintain the circular queue data structure

int p\_s\_head = 0;

int p\_s\_tail = 0;

int s\_ex\_head = 0;

int s\_ex\_tail=0;

//shared buffer

struct readyqueueBUFF rdybuffer[READYQUEUESIZE],exbuffer[READYQUEUESIZE];

// mutex lock for buffer

pthread\_mutex\_t p\_s\_mutex;

pthread\_mutex\_t s\_ex\_mutex;

//semaphores for specifying empty and full indicators

sem\_t p\_s\_emptyBuffers;

sem\_t p\_s\_fullBuffer;

sem\_t s\_ex\_emptyBuffers;

sem\_t s\_ex\_fullBuffer;

//function declaration

void readParameters(const char \*strPara);

void extractFilename(char \*s, char \*d);

void initialize\_locks();

void acceptconnections(int sfd);

void processrequest(char \*req, FILE \*fpout);

void errormode();

void serverfunc(int mode);

void insertinto\_exBUFF(struct readyqueueBUFF \*p,int tail);

void copyfromREADYQueue(struct readyqueueBUFF \*p,int head);

void get\_ex(struct readyqueueBUFF \*p,int head);

void extract\_extension(char \*t,char \*d);

void send\_cat(char cmdtype[4],char path[SIZE],int filesize,int cfd,char clientip[100], time\_t arrtime);

void send\_ls(char cmdtype[4],char path[SIZE],int filesize,int cfd,char clientip[100], time\_t arrtime);

void send\_404(char cmdtype[4],char path[SIZE],int cfd,char clientip[100], time\_t arrtime);

void execute(struct readyqueueBUFF \*p);

void \*consumerfunc(void \*p);

void \*schedulerfunc(void \*p);

void getSJ(struct readyqueueBUFF \*p,int head, int tail);

void \*producerfunc(void \*p);

void insertinto\_buff(struct readyqueueBUFF \*p,int tail);

void logger(char \*clientip,char \*arrt, char \*rcvt, char \*requeststr, char \*requeststatus, int requestsize);

//End declaration

//initialze the locks

void initialize\_locks()

{

pthread\_mutex\_init(&p\_s\_mutex,NULL);

sem\_init(&p\_s\_emptyBuffers,0,READYQUEUESIZE);

sem\_init(&p\_s\_fullBuffer,0,0);

pthread\_mutex\_init(&s\_ex\_mutex,NULL);

sem\_init(&s\_ex\_emptyBuffers,0,READYQUEUESIZE);

sem\_init(&s\_ex\_fullBuffer,0,0);

}

void readParameters(const char \*strPara)

{

char \*tmp=NULL;

FILE \*fproot;

//l :log file

strcat(RootDirectory,"/"); //to append '/' to the rootdirectory

tmp = strstr(strPara, "-l");

if(tmp!=NULL)

{

FILE \*fp;

logmode = 1;

extractFilename(tmp,LogFile);

fp = fopen(LogFile,"a");

if(fp==NULL)

{

fprintf (stderr,"\nERROR: LOGFILE cannot be created or appended\n");

mode=2;

return;

}

fclose(fp);

tmp=NULL;

}

//p: port numner

tmp = strstr(strPara,"-p");

if(tmp!=NULL)

{

char chrport[10];

extractFilename(tmp,chrport);

port=atoi(chrport);

tmp=NULL;

if(port<=1024)

{

fprintf (stderr,"\nERROR: PORT cannot be less than 1024\nRecommended port#: 8080");

mode=2;

return;

}

}

//r: root directory

tmp = strstr(strPara,"-r");

if(tmp!=NULL)

{

FILE \*fp;

extractFilename(tmp,RootDirectory);

strcat(RootDirectory,"/");

tmp=NULL;

fp = fopen(RootDirectory,"r");

if(fp==NULL)

{

fprintf (stderr,"\nERROR: ROOT DIRECTORY DOES NOT EXISTS\n");

mode=2;

return;

}

fclose(fp);

}

//t: time to queue

tmp = strstr(strPara,"-t");

if(tmp!=NULL)

{

char chrtime[10];

extractFilename(tmp,chrtime);

qtime = atoi(chrtime);

tmp=NULL;

if(qtime<0)

{

fprintf (stderr,"\nERROR: Queuing time is incorrect\n");

mode=2;

return;

}

}

//n: e Threads

tmp = strstr(strPara,"-n");

if(tmp!=NULL)

{

char chrThreads[5];

extractFilename(tmp,chrThreads);

eThread = atoi(chrThreads);

tmp=NULL;

if(eThread<=0 || eThread>MAXTHREADS)

{

fprintf (stderr,"\nERROR: Number of threads should be 0<threads<%d\n",eThread=MAXTHREADS);

mode=2;

return;

}

}

//s: Scheduler algo

tmp = strstr(strPara,"-s");

if(tmp!=NULL)

{

char chrSchd[10];

int i=0;

for(i=0;tmp[i]!='\0';i++)

tmp[i] = tolower(tmp[i]);

extractFilename(tmp,chrSchd);

if(strcmp(chrSchd,"sjf")==0)

Schd = 2; //SJF, else the default value

else if (strcmp(chrSchd,"fcfs")==0)

Schd = 1;

else

{

fprintf (stderr,"\nERROR: Please choose between SJF or FCFS scheduling algorithm\n");

mode=2;

return;

}

tmp = NULL;

}

fproot = fopen(RootDirectory,"r");

if(fproot==NULL)

{

fprintf(stderr,"\nERROR: webserver root directory is not accessible/present: %s\n",RootDirectory);

mode=2; return;

}

fclose(fproot);

//h: help

tmp = strstr(strPara,"-h");

if(tmp!=NULL)

{

mode = 2; tmp =NULL; return; //no use setting parameters here, just exit

}

//d: debug mode

tmp = strstr(strPara, "-d");

if(tmp!=NULL)

{

mode = 1; tmp=NULL; eThread =1; qtime=0; Schd = 1; logmode=2; bzero(LogFile,SIZE); return;

}

}

void extractFilename(char \*t,char \*desti)

{

int i =0;

t=t+3; //cross -x and a space

for(i=0;t[i]!=' ';i++)

desti[i]=t[i];

desti[i]='\0';

}

void errormode()

{

printf("\nMYHTTPD USAGE:\n");

fprintf(stderr,"\n-d :Debug mode. Server will accept one connection as a time.");

fprintf(stderr,"\n-h :help");

fprintf(stderr,"\n-l filepath :Log file path. Default is no logging.");

fprintf(stderr,"\n-p port :port. Default is 8080.");

fprintf(stderr,"\n-r dir :set root directory. Default is ~/myhttpd");

fprintf(stderr,"\n-t time :Queuing time to t seconds. Default 60 seconds");

fprintf(stderr,"\n-n threadnum :Set number of threads waiting in the execution thread pool. \n\t\t Default is 4 execution threads.");

fprintf(stderr,"\n-s sched :Set the scheduling policy. It can be either FCFS or SJF. \n\t\t Default will in be FCFS.\n");

}

void logger(char \*clientip,char \*arrt, char \*rcvt, char \*requeststr, char \*requeststatus, int requestsize)

{

if(logmode !=0) //no logging

{

FILE \*fplog;

if(logmode == 1) //logging to a file

{

fplog = fopen(LogFile,"a");

fprintf(fplog,"\n%s - [%s] [%s] \"%s\" %s %d\n",clientip,arrt,rcvt,requeststr,requeststatus,requestsize);

fclose(fplog);

}

else if (logmode == 2) //logging on stdout

{

fprintf(stderr,"\n%s - [%s] [%s] \"%s\" %s %d\n",clientip,arrt,rcvt,requeststr,requeststatus,requestsize);

}

}

}

void serverfunc(int mode)

{

char chrSchd[10];

if(Schd==0 || Schd==1)

strcpy(chrSchd,"FCFS");

else if (Schd==2)

strcpy(chrSchd,"SJF");

if(mode == 0)

printf("\nDAEMON MODE (with settings)\n\n");

else

printf("\nDEBUG MODE (with settings)\n\n");

printf("PORT: %d\nSERVER DEFAULT DIRECTORY: %s\nWAIT BEFORE SCHEDULING TIME: %d\nTHREADS#: %d\nSCHEDULE MODE: %s\nLOGGING MODE: %d\nLOG FILE AT: %s\n\n",

port,

RootDirectory,

qtime,

eThread,

chrSchd,

logmode,

LogFile);

pthread\_t producer\_id, consumer\_id[MAXTHREADS], scheduler\_id;

int i=0;

initialize\_locks();

//Start the producer threads

if (pthread\_create(&producer\_id,NULL,producerfunc,NULL) != 0)

fprintf (stderr, "no producer thread\n");

//Start scheduler thread with timer

if(pthread\_create(&scheduler\_id,NULL,schedulerfunc,NULL)!=0)

fprintf(stderr,"No scheduler thread\n");

//Start the consumer thread(s)

for(i = 0; i < eThread; i++)

if (pthread\_create(&consumer\_id[i],NULL,consumerfunc,NULL) != 0)

fprintf (stderr, "No consumer thread\n");

(void) pthread\_join(producer\_id,NULL);

(void) pthread\_join(scheduler\_id,NULL);

for(i=0;i< eThread;i++)

(void) pthread\_join(consumer\_id[i],NULL);

}

void insertinto\_exBUFF(struct readyqueueBUFF \*p,int tail)

{

//puts("\nEX buffer\n");

exbuffer[tail].markforexecution = p->markforexecution;

exbuffer[tail].filesize = p->filesize;

strcpy(exbuffer[tail].cmdType,p->cmdType);

strcpy(exbuffer[tail].requesttype,p->requesttype);

strcpy(exbuffer[tail].path,p->path);

exbuffer[tail].clientfd = p->clientfd;

exbuffer[tail].arrivaltime = p->arrivaltime;

exbuffer[tail].responsetime = p->responsetime;

strcpy(exbuffer[tail].clientip,p->clientip);

}

void copyfromREADYQueue(struct readyqueueBUFF \*p,int head)

{

p->markforexecution=rdybuffer[head].markforexecution;

p->filesize=rdybuffer[head].filesize;

strcpy(p->cmdType,rdybuffer[head].cmdType);

strcpy(p->requesttype,rdybuffer[head].requesttype);

strcpy(p->path,rdybuffer[head].path);

p->clientfd = rdybuffer[head].clientfd;

p->arrivaltime = rdybuffer[head].arrivaltime;

p->responsetime = rdybuffer[head].responsetime;

strcpy(p->clientip,rdybuffer[head].clientip);

}

void get\_ex(struct readyqueueBUFF \*p,int head)

{

p->markforexecution=exbuffer[head].markforexecution;

p->filesize=exbuffer[head].filesize;

strcpy(p->cmdType,exbuffer[head].cmdType);

strcpy(p->requesttype,exbuffer[head].requesttype);

strcpy(p->path,exbuffer[head].path);

p->clientfd=exbuffer[head].clientfd;

p->arrivaltime=exbuffer[head].arrivaltime;

p->responsetime=exbuffer[head].responsetime;

strcpy(p->clientip,exbuffer[head].clientip);

//puts("\nEXEC: Inserted into FINAL Array\n");

}

void extract\_extension(char \*t,char \*d)

{

int i=0,j=0;

for(i=0;i<=30;i++)

if(t[i]=='.') break;

i++;

for(j=0;t[i]!='\0';i++,j++)

d[j]=t[i];

d[j]='\0';

}

void send\_cat(char cmdtype[4],char path[SIZE],int filesize,int cfd,char clientip[100], time\_t arrtime)

{

char timearr[128],timeresponse[128],timelastmodi[128],c,extn[10],content[100],servername[100],strReq[400];

struct stat statbuff;

FILE \*fp,\*cfp;

int ff;

time\_t now,t\_response;

bzero(strReq,400);

strcat(strReq,cmdtype);

strcat(strReq," ");

strcat(strReq,path);

strcat(strReq," HTTP/1.0");

if(stat(path,&statbuff)!=0)

fprintf (stderr, "STAT Error\n");

gethostname(servername,500);

cfp = fdopen(cfd,"w");

if(cfp == NULL)

fprintf (stderr, "File Error\n");

extract\_extension(path,extn);

if(strcmp(extn,"html")==0)

strcpy(content,"text/html");

else if (strcmp(extn,"gif")==0)

strcpy(content,"image/gif");

else if ((strcmp(extn,"jpeg")==0) || (strcmp(extn,"jpg")==0))

strcpy(content,"image/jpeg");

ff = open(path,O\_RDONLY);

fp = fdopen(ff,"r");

if(fp!=NULL)

{

fprintf(cfp,"HTTP/1.0 200 OK\r\n");

fprintf(cfp,"Server: %s\r\n",servername);

now = time(NULL);

t\_response = now;

strftime(timeresponse, sizeof(timeresponse), "%a, %d %b %Y %H:%M:%S GMT", gmtime(&now));

fprintf(cfp,"Date: %s\r\n",timeresponse);

fprintf(cfp,"Content-Type: %s\r\n",content);

fprintf(cfp,"Content-Length: %d\r\n",filesize);

strftime(timelastmodi, sizeof(timelastmodi), "%a, %d %b %Y %H:%M:%S GMT", gmtime(&statbuff.st\_mtime));

fprintf(cfp,"Last-Modified: %s\r\n",timelastmodi);

fprintf(cfp,"Connection: close\r\n");

fprintf(cfp, "\r\n");

if(strcmp(cmdtype,"GET")==0)

{

int n;

char data[4096];

while ((n = fread(data, 1, sizeof(data), fp)) > 0) fwrite(data, 1, n, cfp);

fclose(fp);

}

strftime(timeresponse, sizeof(timeresponse), "%d/%b/%Y:%H:%M:%S %z", gmtime(&t\_response));

strftime(timearr, sizeof(timearr), "%d/%b/%Y:%H:%M:%S %z", gmtime(&arrtime));

logger(clientip,timearr, timeresponse, strReq, "200", filesize);

}

fflush(cfp);

fclose(cfp);

close(cfd);

close(ff);

}

void send\_ls(char cmdtype[4],char path[SIZE],int filesize,int cfd,char clientip[100], time\_t arrtime)

{

char timearr[128],timeresponse[128],timelastmodi[128],content[100],servername[100],strReq[400];

struct stat statbuff;

struct dirent \*entry;

DIR \*dp;

FILE \*cfp;

time\_t now,t\_response;

bzero(strReq,400);

strcat(strReq,cmdtype);

strcat(strReq," ");

strcat(strReq,path);

strcat(strReq," HTTP/1.0");

stat(path,&statbuff);

gethostname(servername,100);

cfp = fdopen(cfd,"w");

strcpy(content,"text/html");

dp = opendir(path);

if(dp!=NULL)

{

fprintf(cfp,"HTTP/1.0 200 OK\r\n");

fprintf(cfp,"Server: %s\r\n",servername);

now = time(NULL);

t\_response = now;

strftime(timeresponse, sizeof(timeresponse), "%a, %d %b %Y %H:%M:%S GMT", gmtime(&now));

fprintf(cfp,"Date: %s\r\n",timeresponse);

fprintf(cfp,"Content-Type: %s\r\n",content);

fprintf(cfp,"Content-Length: %d\r\n",filesize);

strftime(timelastmodi, sizeof(timelastmodi), "%a, %d %b %Y %H:%M:%S GMT", gmtime(&statbuff.st\_mtime));

fprintf(cfp,"Last-Modified: %s\r\n",timelastmodi);

fprintf(cfp,"Connection: close\r\n");

fprintf(cfp, "\r\n");

fprintf(cfp,"<HTML><BODY><TITLE>DIRECTORY LIST: %s</TITLE><h1>DIRECTORY LIST: %s </h1><table>",path,path);

//

if(strcmp(cmdtype,"GET")==0)

{

struct f

{

char filename[DIR\_SIZE];

}ff[DIR\_SIZE];

char temp[DIR\_SIZE];

int i=0,j=0,len=0,min=0;

while((entry = readdir(dp)))

{

if(!(strcmp(entry->d\_name,".")==0 || strcmp(entry->d\_name,"..")==0))

{

strcpy(ff[i].filename,entry->d\_name);

i++;

}

}

len=i;

for(i=0;i<len;i++) //bubble sort

for(j=0;j<len-i;j++)

if(strcmp(ff[j].filename,ff[j+1].filename)<0)

{

strcpy(temp,ff[j+1].filename);

strcpy(ff[j+1].filename,ff[j].filename);

strcpy(ff[j].filename,temp);

}

while(len!=-1)

{

fprintf(cfp,"<tr><td>%s</td></tr>",ff[len].filename);

len--;

}

fprintf(cfp,"</table></BODY></HTML>");

}

closedir(dp);

fclose(cfp);

}

//for logging

strftime(timeresponse, sizeof(timeresponse), "%d/%b/%Y:%H:%M:%S %z", gmtime(&t\_response));

strftime(timearr, sizeof(timearr), "%d/%b/%Y:%H:%M:%S %z", gmtime(&arrtime));

logger(clientip,timearr, timeresponse, strReq, "200", filesize);

}

void send\_404(char cmdtype[4],char path[SIZE],int cfd,char clientip[100], time\_t arrtime)

{

char timearr[128],timeresponse[128],timelastmodi[128],servername[100],strReq[400];

FILE \*cfp;

time\_t now,t\_response;

bzero(strReq,400);

strcat(strReq,cmdtype);

strcat(strReq," ");

strcat(strReq,path);

strcat(strReq," HTTP/1.0");

gethostname(servername,100);

cfp = fdopen(cfd,"w");

fprintf(cfp,"HTTP/1.0 404 NOT FOUND\r\n");

fprintf(cfp,"Server: %s\r\n",servername);

now = time(NULL);

t\_response = now;

strftime(timeresponse, sizeof(timeresponse), "%a, %d %b %Y %H:%M:%S GMT", gmtime(&now));

fprintf(cfp,"Date: %s\r\n",timeresponse);

fprintf(cfp,"Content-Type: %s\r\n","text/html");

fprintf(cfp,"Connection: close\r\n");

fprintf(cfp, "\r\n");

fprintf(cfp,"404 ERROR NOT FOUND: %s; webserver directory: %s",path,RootDirectory);

fflush(cfp);

fclose(cfp);

//for logging

strftime(timeresponse, sizeof(timeresponse), "%d/%b/%Y:%H:%M:%S %z", gmtime(&t\_response));

strftime(timearr, sizeof(timearr), "%d/%b/%Y:%H:%M:%S %z", gmtime(&arrtime));

logger(clientip,timearr, timeresponse, strReq, "404", 0);

}

void execute(struct readyqueueBUFF \*p)

{

int cfd;

cfd = p->clientfd;

if(strcmp(p->requesttype,"LS")==0)

{

send\_ls(p->cmdType,p->path,p->filesize,cfd,p->clientip,p->arrivaltime);

}

else if(strcmp(p->requesttype,"CAT")==0)

{

send\_cat(p->cmdType,p->path,p->filesize,cfd,p->clientip,p->arrivaltime);

}

else if(strcmp(p->requesttype,"BAD")==0)

{

send\_404(p->cmdType,p->path,cfd,p->clientip,p->arrivaltime);

}

}

void \*consumerfunc(void \*p)

{

struct readyqueueBUFF tmp\_element;

while(1)

{

sem\_wait(&s\_ex\_fullBuffer);

pthread\_mutex\_lock(&s\_ex\_mutex);

//puts("\nEXEC thread: CR Section\n");

get\_ex(&tmp\_element,s\_ex\_head);

s\_ex\_head = (s\_ex\_head + 1) % READYQUEUESIZE;

execute(&tmp\_element);

//puts("EXEC Done");

pthread\_mutex\_unlock(&s\_ex\_mutex);

sem\_post(&s\_ex\_emptyBuffers);

}

}

void \*schedulerfunc(void \*p)

{

while(qtime>0)

{

qtime--;

sleep(1);

}

if(Schd==1 || Schd==0) //FCFS or no scheduling are equal

{

struct readyqueueBUFF tmp\_element;

while (1)

{

//puts("\nSCHD Thread: Entered\n");

sem\_wait(&p\_s\_fullBuffer);

pthread\_mutex\_lock(&p\_s\_mutex);

copyfromREADYQueue(&tmp\_element,p\_s\_head);

p\_s\_head = ( p\_s\_head + 1) % READYQUEUESIZE;

//puts("\nSCHD Thread: Level 1\n");

//starting semaphore for execution

sem\_wait(&s\_ex\_emptyBuffers);

pthread\_mutex\_lock(&s\_ex\_mutex);

insertinto\_exBUFF(&tmp\_element,s\_ex\_tail);

//rdybuffer[tail] = item ;

s\_ex\_tail = (s\_ex\_tail+1) % READYQUEUESIZE;

//printf ("producer: inserted %d \n", item); fflush (stdout);

//puts("\nSCHD Thread: Level 2\n");

pthread\_mutex\_unlock(&s\_ex\_mutex);

sem\_post(&s\_ex\_fullBuffer);

//execution thread semaphore unlocked

pthread\_mutex\_unlock(&p\_s\_mutex);

sem\_post(&p\_s\_emptyBuffers);

//puts("\nSCHD Thread: Level 3\n");

}

}

else if (Schd==2) //SJF

{

struct readyqueueBUFF tmp\_element;

while (1)

{

//puts("\nSCHD Thread algo2: Entered\n");

sem\_wait(&p\_s\_fullBuffer);

pthread\_mutex\_lock(&p\_s\_mutex);

getSJ(&tmp\_element,p\_s\_head,p\_s\_tail);

p\_s\_head = ( p\_s\_head + 1) % READYQUEUESIZE;

//puts("\nSCHD Thread algo2: Level 1\n");

//starting semaphore for execution

sem\_wait(&s\_ex\_emptyBuffers);

pthread\_mutex\_lock(&s\_ex\_mutex);

insertinto\_exBUFF(&tmp\_element,s\_ex\_tail);

//rdybuffer[tail] = item ;

s\_ex\_tail = (s\_ex\_tail+1) % READYQUEUESIZE;

//printf ("producer: inserted %d \n", item); fflush (stdout);

pthread\_mutex\_unlock(&s\_ex\_mutex);

sem\_post(&s\_ex\_fullBuffer);

//execution thread semaphore unlocked

pthread\_mutex\_unlock(&p\_s\_mutex);

sem\_post(&p\_s\_emptyBuffers);

//puts("\nSCHD Thread algo2: Level final\n");

}

}

}

void getSJ(struct readyqueueBUFF \*p,int head, int tail)

{

struct readyqueueBUFF temp, tmp\_arr[READYQUEUESIZE];

int i=0,j=0;

tail--;

if(head==tail);

else if(head<tail)

{

for(i=head;i<=tail;i++)

for(j=head;j<tail;j++)

if(rdybuffer[j].filesize>rdybuffer[j+1].filesize)

{

//temp=arr[j+1];

temp.markforexecution = rdybuffer[j+1].markforexecution;

temp.filesize = rdybuffer[j+1].filesize;

strcpy(temp.cmdType,rdybuffer[j+1].cmdType);

strcpy(temp.requesttype,rdybuffer[j+1].requesttype);

strcpy(temp.path,rdybuffer[j+1].path);

temp.clientfd = rdybuffer[j+1].clientfd;

temp.arrivaltime = rdybuffer[j+1].arrivaltime;

temp.responsetime = rdybuffer[j+1].responsetime;

strcpy(temp.clientip,rdybuffer[j+1].clientip);

//arr[j+1]=arr[j];

rdybuffer[j+1].markforexecution = rdybuffer[j].markforexecution;

rdybuffer[j+1].filesize = rdybuffer[j].filesize;

strcpy(rdybuffer[j+1].cmdType,rdybuffer[j].cmdType);

strcpy(rdybuffer[j+1].requesttype,rdybuffer[j].requesttype);

strcpy(rdybuffer[j+1].path,rdybuffer[j].path);

rdybuffer[j+1].clientfd = rdybuffer[j].clientfd;

rdybuffer[j+1].arrivaltime = rdybuffer[j].arrivaltime;

rdybuffer[j+1].responsetime = rdybuffer[j].responsetime;

strcpy(rdybuffer[j+1].clientip,rdybuffer[j+1].clientip);

//arr[j]=temp;

rdybuffer[j].markforexecution = temp.markforexecution;

rdybuffer[j].filesize = temp.filesize;

strcpy(rdybuffer[j].cmdType,temp.cmdType);

strcpy(rdybuffer[j].requesttype,temp.requesttype);

strcpy(rdybuffer[j].path,temp.path);

rdybuffer[j].clientfd = temp.clientfd;

rdybuffer[j].arrivaltime = temp.arrivaltime;

rdybuffer[j].responsetime = temp.responsetime;

strcpy(rdybuffer[j].clientip,temp.clientip);

}

}

else if(head>tail)

{

int temp\_arr\_size = 0;

int i=0,j=0;

temp\_arr\_size = ((READYQUEUESIZE-1)-head+1) + (tail+1);

//copy to tmp\_arr

for(i=head;i<READYQUEUESIZE;i++)

{

tmp\_arr[i].markforexecution = rdybuffer[i].markforexecution;

tmp\_arr[i].filesize = rdybuffer[i].filesize;

strcpy(tmp\_arr[i].cmdType,rdybuffer[i].cmdType);

strcpy(tmp\_arr[i].requesttype,rdybuffer[i].requesttype);

strcpy(tmp\_arr[i].path,rdybuffer[i].path);

tmp\_arr[i].clientfd = rdybuffer[i].clientfd;

tmp\_arr[i].arrivaltime = rdybuffer[i].arrivaltime;

tmp\_arr[i].responsetime = rdybuffer[i].responsetime;

strcpy(tmp\_arr[i].clientip,rdybuffer[i].clientip);

}

//again copy the remaining elements to tmp\_arr

for(i=0;i<=tail;i++)

{

tmp\_arr[i].markforexecution = rdybuffer[i].markforexecution;

tmp\_arr[i].filesize = rdybuffer[i].filesize;

strcpy(tmp\_arr[i].cmdType,rdybuffer[i].cmdType);

strcpy(tmp\_arr[i].requesttype,rdybuffer[i].requesttype);

strcpy(tmp\_arr[i].path,rdybuffer[i].path);

tmp\_arr[i].clientfd = rdybuffer[i].clientfd;

tmp\_arr[i].arrivaltime = rdybuffer[i].arrivaltime;

tmp\_arr[i].responsetime = rdybuffer[i].responsetime;

strcpy(tmp\_arr[i].clientip,rdybuffer[i].clientip);

}

//now sort tmp\_arr

for(i=0;i<=temp\_arr\_size;i++)

for(j=0;j<temp\_arr\_size;j++)

if(tmp\_arr[j].filesize>tmp\_arr[j+1].filesize)

{

//temp=arr[j+1];

temp.markforexecution = tmp\_arr[j+1].markforexecution;

temp.filesize = tmp\_arr[j+1].filesize;

strcpy(temp.cmdType,tmp\_arr[j+1].cmdType);

strcpy(temp.requesttype,tmp\_arr[j+1].requesttype);

strcpy(temp.path,tmp\_arr[j+1].path);

temp.clientfd = tmp\_arr[j+1].clientfd;

temp.arrivaltime = tmp\_arr[j+1].arrivaltime;

temp.responsetime = tmp\_arr[j+1].responsetime;

strcpy(temp.clientip,tmp\_arr[j+1].clientip);

//arr[j+1]=arr[j];

tmp\_arr[j+1].markforexecution = tmp\_arr[j].markforexecution;

tmp\_arr[j+1].filesize = tmp\_arr[j].filesize;

strcpy(tmp\_arr[j+1].cmdType,tmp\_arr[j].cmdType);

strcpy(tmp\_arr[j+1].requesttype,tmp\_arr[j].requesttype);

strcpy(tmp\_arr[j+1].path,tmp\_arr[j].path);

tmp\_arr[j+1].clientfd = tmp\_arr[j].clientfd;

tmp\_arr[j+1].arrivaltime = tmp\_arr[j].arrivaltime;

tmp\_arr[j+1].responsetime = tmp\_arr[j].responsetime;

strcpy(tmp\_arr[j+1].clientip,tmp\_arr[j].clientip);

//arr[j]=temp;

tmp\_arr[j].markforexecution = temp.markforexecution;

tmp\_arr[j].filesize = temp.filesize;

strcpy(tmp\_arr[j].cmdType,temp.cmdType);

strcpy(tmp\_arr[j].requesttype,temp.requesttype);

strcpy(tmp\_arr[j].path,temp.path);

tmp\_arr[j].clientfd = temp.clientfd;

tmp\_arr[j].arrivaltime = temp.arrivaltime;

tmp\_arr[j].responsetime = temp.responsetime;

strcpy(tmp\_arr[j].clientip,temp.clientip);

}

//now move the elements to rdyqueue

for(i=head;i<READYQUEUESIZE;i++)

{

rdybuffer[i].markforexecution = tmp\_arr[i].markforexecution;

rdybuffer[i].filesize = tmp\_arr[i].filesize;

strcpy(rdybuffer[i].cmdType,tmp\_arr[i].cmdType);

strcpy(rdybuffer[i].requesttype,tmp\_arr[i].requesttype);

strcpy(rdybuffer[i].path,tmp\_arr[i].path);

rdybuffer[i].clientfd = tmp\_arr[i].clientfd;

rdybuffer[i].arrivaltime = tmp\_arr[i].arrivaltime;

rdybuffer[i].responsetime = tmp\_arr[i].responsetime;

strcpy(rdybuffer[i].clientip,tmp\_arr[i].clientip);

}

j=i; //important

for(i=0;i<=tail;i++,j++)

{

rdybuffer[i].markforexecution = tmp\_arr[j].markforexecution;

rdybuffer[i].filesize = tmp\_arr[j].filesize;

strcpy(rdybuffer[i].cmdType,tmp\_arr[j].cmdType);

strcpy(rdybuffer[i].requesttype,tmp\_arr[j].requesttype);

strcpy(rdybuffer[i].path,tmp\_arr[j].path);

rdybuffer[i].clientfd = tmp\_arr[j].clientfd;

rdybuffer[i].arrivaltime = tmp\_arr[j].arrivaltime;

rdybuffer[i].responsetime = tmp\_arr[j].responsetime;

strcpy(rdybuffer[i].clientip,tmp\_arr[j].clientip);

}

}

p->markforexecution = rdybuffer[head].markforexecution;

p->filesize = rdybuffer[head].filesize;

strcpy(p->cmdType,rdybuffer[head].cmdType);

strcpy(p->requesttype,rdybuffer[head].requesttype);

strcpy(p->path,rdybuffer[head].path);

p->clientfd = rdybuffer[head].clientfd;

p->arrivaltime = rdybuffer[head].arrivaltime;

p->responsetime = rdybuffer[head].responsetime;

strcpy(p->clientip,rdybuffer[head].clientip);

/\*for(i=0;i<3;i++)

printf("\nFilesize:(%d) %d",i,rdybuffer[i].filesize);\*/

}

void insertinto\_buff(struct readyqueueBUFF \*p,int tail)

{

fflush(stdout);

rdybuffer[tail].markforexecution = p->markforexecution;

rdybuffer[tail].filesize = p->filesize;

strcpy(rdybuffer[tail].cmdType,p->cmdType);

strcpy(rdybuffer[tail].requesttype,p->requesttype);

strcpy(rdybuffer[tail].path,p->path);

rdybuffer[tail].clientfd = p->clientfd;

rdybuffer[tail].arrivaltime = p->arrivaltime;

rdybuffer[tail].responsetime = p->responsetime;

strcpy(rdybuffer[tail].clientip,p->clientip);

}

void \*producerfunc(void \*p)

{

int cfd,a=0,b= sizeof(int),i=0;

char requestblk[SER\_BUFF\_SIZE];

char out[SIZE],cmdType[4],filerequest[SIZE],httpversion[100],hostname[250], \*clientipstr=NULL;

mode\_t modes;

FILE \*fpin,\*fp;

unsigned int len;

time\_t now;

struct readyqueueBUFF tmp\_element;

struct stat statbuf;

struct sockaddr\_in ser,c;

struct hostent \*hp;

gethostname(hostname,250);

hp = gethostbyname(hostname);

sfd=socket(AF\_INET,SOCK\_STREAM,0);

ser.sin\_family=AF\_INET;

ser.sin\_addr.s\_addr=htonl(INADDR\_ANY);

ser.sin\_port=htons(port);

len=sizeof(ser);

if((bind(sfd,(struct sockaddr \*)&ser,len))==-1)

{

fprintf (stderr, "SOCKET BIND ERROR\n");

exit(1);

}

if(listen(sfd,1)==-1) //started listening

{

fprintf (stderr, "SOCKET LISTEN ERROR\n");

exit(1);

}

while(1)

{

struct sockaddr\_in c;

FILE \*fpout;

len = sizeof(c);

strcpy(tmp\_element.cmdType,"");

tmp\_element.filesize=0;

strcpy(tmp\_element.path,"");

strcpy(tmp\_element.requesttype,"");

tmp\_element.arrivaltime = 0;

//printf("hello %d",sfd);

if((cfd=accept(sfd,(struct sockaddr \*)&c,&len))!=-1)

fpin = fdopen(cfd,"r");

//fpout = fdopen(cfd,"w");

//fprintf(fpout,"gggg\r\n");

//fflush(fpout);

fgets(requestblk,SER\_BUFF\_SIZE,fpin);

//fclose(fpin);

//puts(requestblk);

sscanf(requestblk,"%s %s %s",&cmdType,&filerequest,&httpversion);

//printf("\n%s %s %s\n",cmdType,filerequest,httpversion);

//memset for structure

//strcpy(filerequest,"/root/index.html");

tmp\_element.arrivaltime = time(&now);

strcpy(tmp\_element.cmdType,cmdType);

clientipstr=inet\_ntoa(c.sin\_addr);

strcpy(tmp\_element.clientip,clientipstr);

tmp\_element.clientfd = cfd;

if(filerequest[0] == '.')

{

strcpy(tmp\_element.requesttype,"BAD");

tmp\_element.filesize=0;

strcat(tmp\_element.path,filerequest);

}

else

{

int i;

char \*s = (char\*)getenv("HOME"); //fetch

if(filerequest[0] == '~')

{

strcat(tmp\_element.path,s);

strcat(tmp\_element.path,RootDirectory);

for(i=0;i<strlen(filerequest);i++)

filerequest[i]=filerequest[i+2];

filerequest[i]='\0';

}

else if (filerequest[1] == '~')

{

strcat(tmp\_element.path,s);

strcat(tmp\_element.path,RootDirectory);

for(i=0;i<strlen(filerequest);i++)

filerequest[i]=filerequest[i+3];

filerequest[i]='\0';

}

else

strcat(tmp\_element.path,RootDirectory);

strcat(tmp\_element.path,filerequest);

if(stat(tmp\_element.path,&statbuf)!=0)

{

strcpy(tmp\_element.requesttype,"BAD");

tmp\_element.filesize=0;

}

else

{

modes=statbuf.st\_mode;

if(S\_ISDIR(modes))

{

char tmpFile[SIZE]; FILE \*fp;

strcpy(tmpFile,tmp\_element.path);

strcat(tmpFile,"/index.html");

fp=fopen(tmpFile,"r");

if(fp!=NULL) //file found

{

strcpy(tmp\_element.path,tmpFile);

strcpy(tmp\_element.requesttype,"CAT");

}

else

{

strcpy(tmp\_element.requesttype,"LS");

}

stat(tmp\_element.path,&statbuf);

tmp\_element.filesize = statbuf.st\_size;

}

else

{

strcpy(tmp\_element.requesttype,"CAT");

tmp\_element.filesize = statbuf.st\_size;

}

}

}

//puts("\nPROD: PARSED ");

//printf("%d %s %s %s\n",tmp\_element.filesize,tmp\_element.requesttype,tmp\_element.path,tmp\_element.cmdType);

//semaphore for readyqueue

sem\_wait(&p\_s\_emptyBuffers);

pthread\_mutex\_lock(&p\_s\_mutex);

insertinto\_buff(&tmp\_element,p\_s\_tail);

//puts("\nPROD: Level 1\n");

//rdybuffer[tail] = item ;

p\_s\_tail = (p\_s\_tail+1) % READYQUEUESIZE;

//printf ("producer: inserted %d \n", item); fflush (stdout);

pthread\_mutex\_unlock(&p\_s\_mutex);

sem\_post(&p\_s\_fullBuffer);

//semaphore unlocked

}

}

int main(int argc, char \*argv[])

{

char args[ARGSIZE];

int i=0;

pid\_t pid;

for(i=0;i<argc;i++)

{

strcat(args,argv[i]);

strcat(args," ");

}

readParameters(args);

switch (mode)

{

case 2: errormode(); break;

case 1: serverfunc(mode); break; //single mode

case 0: pid = fork(); if(pid==0) {serverfunc(mode); break;} //daemon mode

}

return 0;

}

## Makefile

default:

gcc source.c -o myhttpd -pthread