# **Design Document for Multi-threaded Web Server**

Team G20

Gaoqi Luo 50026256 [gaoqiluo@buffalo.edu](mailto:gaoqiluo@buffalo.edu)

Yan Ma 50062112 [yanma@buffalo.edu](mailto:yanma@buffalo.edu)

Aditi Madan 50062113 [aditimad@buffalo.edu](mailto:aditimad@buffalo.edu)

1. **Overview**

The objective of this project is to implement a multithreaded web server “myhttpd” in C/C++ on a UNIX-based platform.

1. **Responsibilities of each team member in the project**

|  |  |
| --- | --- |
| Gaoqi Luo | Build up the server (socket communication, thread pool initiation, scheduler thread initiation, synchronizations), SJF scheduler, Http request analyze, file directory path analyze, generate index page. |
| Yan Ma | -d set debug mode, FCFS scheduler, produce the logging information, respond-head generation. |
| Aditi Madan | -p set listening port, -t set queuing time, -n threadnum, -h print command information, LastModifyTime() function, Time() function. |

1. **The data structures**

(used for the implementation of the queuing, scheduling, multithreading and synchronization components of the server)

Ready-queue: C++ queue – “vector<Request\*> ready\_queue”, used for save the incoming http request.

Request: “struct request”

struct Request{

char\* req\_head;

char\* ip;

char\* recv\_time;

char\* asgn\_time;

int status;

int client\_socket;

int requestType;

const char\* requestDir;

int contentLength

char\* contentType;

int isDir;

};

Thread-pool: struct array - “struce exec\_thread \* thread\_pool”, used for maintaining “threadnum” exec\_threads, and wait for scheduler’s selection.

struct exec\_thread{

pthread\_t thread;

struct Request\* req;

sem\_t \*sem;

int isFree;

};

Lock: there are two locks, one is for synchronizing the request for the thread\_pool, another is for synchronizing the request for ready queue.

pthread\_mutex\_t queue\_lock;

pthread\_mutex\_t lock;

Semaphore: there are three different semaphores. One is used by scheduler to manage incoming requests, one is used for choosing the thread, another one is initiated in each thread.

sem\_t s\_queue;

sem\_t s\_threads;

sem \*sem;

Conditional variable: this cv is used for synchronizing the main thread and the initiated thread in the thread\_pool .

pthread\_cond\_t cv;

1. **Context switches**

(How are context switches between threads implemented in our code)

First, we need to switch between the main thread and the initiated thread, this is because we need to create thread one by one to make sure each newly created thread has its correct thread ID. Here we use a semaphore and a conditional variable, to ensure the thread switches.

Context switch will also happen when a schedule thread is informed to choose a thread to manage the incoming request. When there is any incoming request, sem\_wait(s\_queue) will be triggered, and the scheduler will change its state from “wait” to “run”. There is another situation that the schedule thread has to change its state from “run” to “wait” - no free thread to choose, so the scheduler just waits until there is any thread claims free and sem\_post(s\_thread), and then changes its state from “wait” to “run”.

Context switches will happen frequently on the threads in the thread\_pool too. Every time, any idle thread is chosen by the scheduler, it will have to change its state from “wait” to “run”, and every time any thread that finished its job will change its state from “run” to “wait”.

1. **Race conditions avoided**

(How are your race conditions avoided in your code?)

Race conditions will happen when different threads acquire and want to modify the same-shared resources. In this project, there are several global shared resources, such as ready-queue, thread\_pool. For ready-queue, main thread and schedule thread may visit it at the same time, so we need a lock to realize mutual exclusive. For thread\_pool, when initiating the newly created thread, main thread may change the content of the thread\_pool, so we need a lock too.

Race condition will not occur when different threads only need to read the shared resources, such as when certain work thread is reading its assigned request, scheduler is searching for idle thread at the same time.

1. **Critique our design**

(Briefly critique your design, pointing out advantages and disadvantages in your design choices.)

|  |  |
| --- | --- |
| Advantages | Excellent synchronization between different threads.  Correctly identify different type of file path. |
| Defects | After starting the server in debug mode, there isn’t any efficent way to close the background thread when shut down the server.  Implement only part of Http/1.0 protocal. |

1. **Online or offline resources we consulted**

(Please cite any online or offline resources you consulted while preparing your project, other than the course materials.)

<http://www.linuxhowtos.org/C_C++/socket.htm>

<http://softpixel.com/~cwright/programming/threads/threads.c.php>

<http://www.enderunix.org/docs/eng/daemon.php>

<http://www.w3schools.com/html/default.asp>

Attachment 1: Readme

1.Type "make" to compile all the source files.

2.Type "./myhttpd [-options]..." to start the server.

3. Using web browser to access the server,firefox is recommanded.

4.All the functions according to the project specification are implemented.

5.There may be some bugs I hadn't discovered yet.

Attachment 2: Source code

1. main.cpp

#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <iostream>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <string>

#include <cstring>

#include <dirent.h>

#include <pwd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <pthread.h>

#include "head.h"

#include <vector>

#include <semaphore.h>

using namespace std;

/\*\*

myhttpd web sever

\*\*/

//thread pool

struct exec\_thread\* threadpool;

//scheduler

pthread\_t scheduler;

//request queue

vector<Request\*> ready\_queue;

pthread\_mutex\_t queue\_lock;//for request ready queue

pthread\_mutex\_t lock;

sem\_t sem\_q;//for schedduler start

sem\_t sem\_ths;//for scheduler to choose exec\_thread

pthread\_cond\_t cv;

int debug=0;

int port=8080;//default port

char\* logFile=NULL;

char\* sDir=NULL;

int sleep\_time=60; //scheduler sleep time

int thread\_num=4;//default thread\_num in the thread\_pool

int sche\_flag=1;//default scheduler

int main(int argc,char\*\* argv){

opterr=0;

int options;

while((options=getopt(argc,argv,"l:p:r:t:n:s:dh"))!=-1){

switch(options){

case'd':

{

debug=1;

thread\_num=1;

break;

}

case'h':

{

Info();

//print options then exit

exit(0);

break;

}

case'r':

{

sDir=optarg;

break;

}

case't':

{

sleep\_time=atoi(optarg);

break;

}

case'l':

{

logFile=optarg;

break;

}

case'p':

{

port=atoi(optarg);

break;

}

case'n':

{

thread\_num=atoi(optarg);

break;

}

case's':

{

string sche=string(optarg);

if(sche=="SJF"){

sche\_flag=2;

}

else if(sche == "FCFS"){

sche\_flag = 1;

}

else{

printf("Unknown schedule algorithm!\n");

exit(0);

}

break;

}

default:

printf("Unknown Options!");

}

}

//deamonize

if(!debug){

pid\_t pid, sid;

if ((pid=fork())<0){

printf("eroor\n");

exit(0);

}else if (pid!=0) {

printf("deamonize thread pid = %d\n",pid);

fflush(stdout);

exit (0);

}

umask(0);

sid = setsid();

if(sid < 0){

printf("set sid fail\n");

exit(0);

}// session leader

if(chdir("./")<0){

printf("change chdir fail\n");

exit(0);

}

}

//socket starts

struct sockaddr\_in saddr;

int server\_socket,client\_socket;

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

if(server\_socket==-1){

printf("socket creating error...\n");

exit(0);

}

saddr.sin\_family=AF\_INET;

saddr.sin\_addr.s\_addr=INADDR\_ANY;

saddr.sin\_port=htons(port);

bzero(&(saddr.sin\_zero),8);

if(bind(server\_socket,(struct sockaddr\*)&saddr,sizeof(saddr))==-1){

printf("binding error: this port is in use!\n");

exit(0);

}

if(listen(server\_socket,20)==-1){ printf("listen error...\n");

exit(0);

}

threadpool=new exec\_thread[thread\_num];

int i;

pthread\_cond\_init(&cv,NULL);

pthread\_mutex\_init(&lock,NULL);

pthread\_mutex\_init(&queue\_lock,NULL);

sem\_init(&sem\_q,0,0);

sem\_init(&sem\_ths,0,thread\_num);

//create threads

for(i=0;i<thread\_num;i++){

//the locker here is used for synchronize the threadId

pthread\_mutex\_lock(&lock);

pthread\_create(&(threadpool[i].thread),NULL,run,&i);

threadpool[i].isFree=1;

pthread\_cond\_wait(&cv,&lock);

pthread\_mutex\_unlock(&lock);

}

//create schedule thread

if(sche\_flag==2){

pthread\_create(&scheduler,NULL,SJF\_scheduler,NULL);

}else{

pthread\_create(&scheduler,NULL,FCFS\_scheduler,NULL);

}

printf("Server started...\n");

fflush(stdout);

//accept income connections

char buffer[1024];

while(1){

client\_socket=accept(server\_socket,NULL,NULL);

if(client\_socket==-1){

printf("client socket error...");

}

char\* recv\_time=Time();//get receive time for dubug mode and logging

fflush(stdout);

struct Request \*req;

recv(client\_socket,buffer,1024,0);

fflush(stdout);

req=http\_analyzer(buffer);

req->client\_socket=client\_socket;

//if in debug model or logfile activated,get the client address and recv\_time

if(debug==1||logFile){

socklen\_t r\_length = sizeof(struct sockaddr\_in);

struct sockaddr\_in r;

req->recv\_time=recv\_time;//record receive time

if(getpeername(client\_socket, (struct sockaddr\*)&r, &r\_length) == 0){

//get ip addr

req->ip = inet\_ntoa(r.sin\_addr);

}

}

pthread\_mutex\_lock(&queue\_lock);

ready\_queue.push\_back(req);//push the request into the request ready queue

sem\_post(&sem\_q);//one request is accepted, scheduler can start

pthread\_mutex\_unlock(&queue\_lock);

}

}

/\*\*execution thread function\*\*/

void\* run(void\* id){

pthread\_mutex\_lock(&lock);

int thread\_id=\*(int\*)id;

//sem\_t \*sem;

//in MAC OS, only sem\_open can be used

/\*

if((sem =sem\_open("sem",O\_CREAT,0,0))== NULL){

cout<<"Thread ["<<thread\_id<<"] semephore init failed..."<<endl;

}

else{

threadpool[thread\_id].sem= \*sem;

}

\*/

if(sem\_init(&(threadpool[thread\_id].sem),0,0)!=0){

printf("Thread %d semephore init failed...",thread\_id);

}

pthread\_cond\_signal(&cv);//after using the pointer "id",wake up main thread.

pthread\_mutex\_unlock(&lock);//after

char buffer[1024];

//keep looking for job

while(true){

sem\_wait(&(threadpool[thread\_id].sem));//wait for scheduling

if(threadpool[thread\_id].isFree==0){//this thread is scheduled

struct Request\* r=threadpool[thread\_id].req;

if((r->contentLength>0)){//requested file is exsit,just response

string rHead;

printf("response....\n");

r->status=200;//200 - find file

getResponse(r,rHead);

//Head Response just send back response

if(r->requestType==1){

if(r->isDir==1){

//if it is a directory then generate fileindex page

char\* dir\_index=generateIndex(r);//generate html first so that the actrual contentlength can be modified

// response head

const char\*res=rHead.c\_str();

send(r->client\_socket,res,strlen(res),0);

//generate fileindex html page

send(r->client\_socket,dir\_index,strlen(dir\_index),0);

free(dir\_index);

}

else{

//send file

const char\*resp=rHead.c\_str();

send(r->client\_socket,resp,strlen(resp),0);

int lenth=1;

/\*\*send file data\*\*/

int fh=open(r->requestDir,O\_RDONLY);

while(lenth>0){

lenth=read(fh,buffer,1024);

if(lenth>0){

send(r->client\_socket,buffer,lenth,0);//continueous sending the data

//sleep(1000);

}

}

close(fh);

}

}

else if(r->requestType==2){

const char\* resp=rHead.c\_str();

send(r->client\_socket,resp,strlen(resp),0);

//Get Response send back response, and open the requested file

}

}else{

//request not found

r->status=404;

string rHead;

getResponse(r,rHead);//404 NOT FOUND

const char\* resp=rHead.c\_str();

send(r->client\_socket,resp,strlen(resp),0);

}

//close the socket

close(r->client\_socket);

if(logFile!=NULL||debug==1){

logging(r,debug,logFile);

}

threadpool[thread\_id].isFree=1;

free(r);

sem\_post(&sem\_ths);//this thread is free, notify the scheduler

}

}

return NULL;

}

//analyse http request

struct Request\* http\_analyzer(char\* p){

char split\_line[]="\n";

int flag=0;

struct Request\* req=(struct Request\*)(malloc(sizeof(struct Request)));

char\* m=strtok(p,split\_line);

if(debug==1||logFile){//if in debug model or logfile activated,record the first line of request

char\* n = (char\*)malloc(sizeof(char)\*strlen(m));

int len = strlen(m);

strncpy(n,m,len-1);//get rid of "\n" in the last of string

\*(n+len-1)='\0';//add end

req->req\_head=n;

}

char split\_word[]=" \n";

char\* tok=strtok(p,split\_word);

while(tok!=NULL){

if(flag==0){

//first tok is about request method

string head=string(tok);

if(head=="HEAD"){

req->requestType=2;

}

else if(head=="GET"){

req->requestType=1;

}

}

//file directory of request

else if(flag==1){

char\* dir = (char\*)malloc(sizeof(char)\*strlen(tok));

strcpy(dir,tok);//file directory

//analyse the file path

analyseUrl(dir,sDir,req);

if(req->isDir==0){

//if request a file,get the file length

ifstream instream;

instream.open(req->requestDir,ios::in|ios::binary);

if(!instream){

req->contentLength=-1;

}else{

instream.seekg(0,ios::end);

int f\_len=instream.tellg();

//contentlength can be used in SJF, or under debug mode, or log,or decide wether a file is find

req->contentLength=f\_len;

}

instream.close();

}else if(req->isDir==1){

//check if the diretory exists

DIR \*direcroy=NULL;

direcroy=opendir(req->requestDir);

if(direcroy==NULL){

req->contentLength=-1;

}

else{

//if direcory is found,then set the contentlength to 200 to define it

req->contentLength=200;

}

}else{//unknow Request

req->contentLength=-1;

}

}

else if(flag==3){

//dont need other rquest information, just break the loop

if(string(tok)!="HTTP/1.0"||string(tok)!="HTTP/1.1"){

req->contentLength=-1;

}

break;

}

flag++;

tok=strtok(NULL,split\_word);

}

return req;

}

/\*\*SJF\*\*/

void\* SJF\_scheduler(void\* p){

sleep(sleep\_time);

while(true){

sem\_wait(&sem\_q);

if(ready\_queue.size()!=0){

/\*\*get reqeust from the ready queue\*\*/

int index=0,i=0;//to store the index of the shortest request

struct Request\* req;

pthread\_mutex\_lock(&queue\_lock);//lock

//the first job is the default job to chose to run,there is atleast 1 job in the queue,when scheduler is waked up

req =ready\_queue[0];

//if(ready\_queue.size()==1){

// ready\_queue.erase(ready\_queue[0]);

//}

//find the shortest job

for(unsigned int i=0;i<ready\_queue.size();i++){

if(req->contentLength > ready\_queue[i]->contentLength){

req=ready\_queue[i];

index=i;

}

}

//after getting the shortest job in the queue,then delete it

ready\_queue.erase((ready\_queue.begin()+index));

pthread\_mutex\_unlock(&queue\_lock);

/\*\*assign the request to a certain execution thread\*\*/

sem\_wait(&sem\_ths);//wait if there's no idle threads,actually we can spin here too.

for(i=0;i<thread\_num;i++){

if(threadpool[i].isFree==1){

if(debug==1||logFile){

//record the assign time

req->asgn\_time=Time();

}

threadpool[i].isFree=0;

threadpool[i].req=req;

//wake up the free thread.

sem\_post(&(threadpool[i].sem));

break;

}

}

}

}

return NULL;

}

/\*\*FCFS\*\*/

void\* FCFS\_scheduler(void\* p){

cout<<"Default scheduler: FCFS, starts after "<<sleep\_time<<"seconds!<<<<"<<endl;

sleep(sleep\_time);

while(true){

sem\_wait(&sem\_q);

if(ready\_queue.size()!=0){

int i=0;

//when access to the ready queue, we need a lock

pthread\_mutex\_lock(&queue\_lock);

//get the head of the queue,FCFS,then delete it

struct Request\* req =ready\_queue[0];

ready\_queue.erase(ready\_queue.begin());

//dont forget to free the lock

pthread\_mutex\_unlock(&queue\_lock);

//there are only 4 threads in the pool,so use a sem to control

//if there no free thread, just wait

sem\_wait(&sem\_ths);

for(i=0;i<thread\_num;i++){

if(threadpool[i].isFree==1){

if(debug == 1||logFile){

//record the assgn time

req->asgn\_time=Time();

}

threadpool[i].isFree=0;

threadpool[i].req=req;

//wake up the selected thread

sem\_post(&(threadpool[i].sem));

break;

}

}

}

}

return NULL;

}

//parse the file path

void analyseUrl(char\* path,char\* sDir,struct Request\* r){

char\* dir=NULL;

char\* type=(char\*)malloc(sizeof(char)\*9);

if(strlen(path)==1){//only "/" after port,such as "8080/"

r->isDir=1;//it indicate a dir

if(sDir==NULL){

//server local dir

dir=(char\*)malloc(sizeof(char)\*2);

//current dir

strcpy(dir,"./");

}else{

dir=sDir;

}

//return 1;//is directory

}else if(!sDir){// no ~, no sDir

//get the file path after "/"

dir=1+path;

}else if(\*(path+1)=='~'){

string new\_dir;

//get the server homedir path

struct passwd\* pw;

pw=getpwuid(getuid());

//get pw info,including home directory

if(strlen(path)==2){

//if there no further path after the /~, then return homedir+/myhtpd/,as mentionned in PROTOCOL part

new\_dir=string(pw->pw\_dir)+"/myhttpd/";

}else{

//else append the rest path

new\_dir=string(pw->pw\_dir)+"/myhttpd/"+string(path+2);

}

dir=(char\*)malloc(sizeof(char)\*new\_dir.size());

//printf("%s\n",dir);

strcpy(dir,new\_dir.c\_str());

}else{//server directory sepecified

string sDir=string(path+1)+string(sDir);

dir=(char\*)malloc(sizeof(char)\*sDir.size());

strcpy(dir,sDir.c\_str());

}

//9 means the max length of the string of content-type: "test/html" or "image/..."

//do analyse the type

if(\*(dir+strlen(dir)-1) == '/'){

//if it is derecory, try to find index.html in this dir

struct stat\* buff=(struct stat\*)malloc(sizeof(struct stat));

string f\_path= string(dir)+"index.html";

//stat(), similar to the open() to check wether certain file is exit or not, the first argument is the filepath

int result = stat(f\_path.c\_str(),buff);

if(result ==0){

char\* f= (char\*)malloc(strlen(f\_path.c\_str())\*sizeof(char));

strcpy(f,f\_path.c\_str());

r->isDir=0;

r->requestDir=f;

free(dir);

}else{

//index.html not found , set the direcory, finally we will see a fileindex page

r->requestDir=dir;

r->isDir=1;

}

strcpy(type,"text/html");

r->contentType=type;

free(buff);

//contentType has no sense when the path indicates a dir and there is no index.html, but anyway, just set it to 'test/html'

}else{

//if request a file, it can be text or some other type file, or image

r->requestDir=dir;

r->isDir=0;

char\* tep = (char\*)malloc(sizeof(char)\*strlen(dir));

strcpy(tep,dir);

char\* tok=strtok(tep,".");

//find the suffix

tok=strtok(NULL,".");

//file without suffix will be ignored

if(tok==NULL){

free(tep);

r->isDir=-1;

return;

}

string s=string(tok);

if(s=="gif"||s=="png"||s=="bmp"||s=="jpg"){

string step="image/"+s;

strcpy(type,step.c\_str());

}

else if(s=="htm"||s=="html"||s=="txt"){

strcpy(type,"text/html");

}

free(tep);

r->contentType=type;

}

}

1. options.cpp

#include "head.h"

#include <time.h>

#include <iostream>

#include <fstream>

#include <sys/stat.h>

#include <cstring>

#include <dirent.h>

using namespace std;

//last modify time

char\* lastmodifyTime(const char\* dir ){

char\* mtime=(char\*)malloc(sizeof(char)\*25);

struct stat\* buff=(struct stat\*)malloc(sizeof(struct stat));

stat(dir,buff);

////

//time\_t st\_mtime in struct \*stat identifies the last modify time,

//ctime: convert a time\_t type time into string

mtime=ctime(&(buff->st\_mtime));

return mtime;

}

/\*\*HTTP Response Head

include:

status,date,server,last-modified,content-type,content-length

this should be sent back with the socket

\*\*/

void getResponse(struct Request\* q,string &response){

string server="Gaoqi's Web Server.\n";

string date="Date:"+string(Time())+"\n";

string s;

//successful request

switch(q->status){

case 404:

s="HTTP/1.0 404 NOT FOUND\n";

response=s+date+server+"\n<html><h1>File Not Found</h1></html>";

break;

case 200:

{

s="HTTP/1.0 200 OK\n";

char con\_len[10];

string con\_Type="Content-Type:"+string(q->contentType)+"\n";

sprintf(con\_len,"%d",q->contentLength);

//the last head line should be separated by a blank line with the content.

string conLength="Content-Length:"+string(con\_len)+"\n\n";

string lasdMod="Last-Modified:"+string(lastmodifyTime(q->requestDir));

response=s+date+server+lasdMod+con\_Type+conLength;

}

break;

}

}

/\*\*

current time

\*\*/

char\* Time(){

time\_t timet;

timet=time(NULL);

char \*format\_time=(char\*)malloc(sizeof(char)\*30);

struct tm\* p;

p=localtime(&timet);

strftime(format\_time,30,"[%d/%b/%Y:%T %z]",p);

return format\_time;

}

/\*\*Generate Log information

when in the debug mode, print log information

when there is a logfile, then wirte the logfile

logging info:

IP,receive time,execution time,request head,status,content-length

\*\*/

void logging(struct Request\* r,int debug,char\* ldir){

string stat;

//http 1.0

switch(r->status){

case 404:

//bad request

stat="404";

break;

case 200:

//find file

stat="200";

break;

default:

stat="500";

}

if(debug==1){

//print log information

cout<<r->ip<<" - "<<r->recv\_time<<" "<<r->asgn\_time<<" \""<<r->req\_head<<"\" "<<stat<<" "<<r->contentLength<<endl;

}else{

//there is a logfile

string log;

char content\_lenth[10];

sprintf(content\_lenth,"%d",r->contentLength);

//write logfile

log=string(r->ip)+" - "+string(r->recv\_time)+" "+string(r->asgn\_time)+" \""+string(r->req\_head)+"\" "+stat+" "+string(content\_lenth)+"\n";

ofstream outstream;

outstream.open(ldir,ios::out|ios::app);

outstream.write(log.c\_str(),log.size());

outstream.close();

}

}

void Info(){

cout<<"SERVER NAME\n"

<<" myhttpd\n\n"

<<"SYNOPSIS\n"

<<" myhttpd [-d] [-h] [-p port] [-r dir] [-t time] [-n threadnum] [-s sched]\n\n"

<<"DESCRIPTION"

<<" myhttpd is a simple web server.It binds to a given port on the given address and waits for incoming HTTP/1.0 requests.It serves content from the given directory.That is, any requests for documents is resolved relative to this directory.\n\n"

<<" -d\n"

<<" :Enter debugging mode.That is ,the server process will not daemonize,only accept one connection at a time and enable logging to stdout. Without this option, the web server runs as a daemon process in the background.\n\n"

<<" -h\n"

<<" :Print the usage of all the options and exit.\n\n"

<<" -l file\n"

<<" :Log all requests to the given file.\n\n"

<<" -p port\n"

<<" :Listen on the given port. If not provided, 8080 will be the default port.\n\n"

<<" -r dir\n"

<<" :Set the root directory for the http server to dir. \n\n"

<<" -t time\n"

<<" :Set the queuing time to time seconds. 60s is the default time\n\n"

<<" -n threadnum\n"

<<" :Set number of threads waiting ready in the execution thread pool to threadnum==4 by default.\n\n"

<<" -s sched\n"

<<" :Set the sheduling policy. It can be either CFS or SJF. The default will be FCFS.\n";

}

/\*\*

when cant find the index.html in the directory,

then just generate fileindex html page for direcory

subdirectory is append with '/'

all the files and directories have their 'herf' attributes

\*\*/

char\* generateIndex(struct Request\* r){

string html="<html><p>Direcory:</p><table width=\"100%\" style=\"border:solid 1px #fff000;\"><tr><th>FILE</th><th>LAST MODIFIED TIME</th></tr>";

struct dirent \*\*file\_name=NULL;

int file\_num;

file\_num=scandir(r->requestDir,&file\_name,0,alphasort);

if(file\_num<0){

html=html+"</table><p>empty directory!</p> </html>";

}else{

for(int i=0;i<file\_num;i++){

//skip hidden file

if(!file\_name[i]->d\_name){

html=html+"<p>empty directory!</p>";

break;

}

if(file\_name[i]->d\_name[0] =='.'){

continue;

}

struct stat buff;

string subdir=string(r->requestDir)+string(file\_name[i]->d\_name);

lstat(subdir.c\_str(),&buff);

if(S\_ISDIR(buff.st\_mode)){

//if it is directory,add '/'

html=html+"<tr><th><a href=\""+string(file\_name[i]->d\_name)+"/\">"+string(file\_name[i]->d\_name)+"</a></th>";

}else{

//else if it is a file

html=html+"<tr><th><a href=\""+string(file\_name[i]->d\_name)+"\">"+string(file\_name[i]->d\_name)+"</a></th>";

}

html=html+"<th>"+string(lastmodifyTime(subdir.c\_str()))+"</th></tr>";

}

html=html+"</table></html>";

}

//r->contentLength=html.size();//modify the content length

char\* buffer=(char\*)malloc(html.size()\*sizeof(char));

strcpy(buffer,html.c\_str());

return buffer;

}

1. head.h

#include <pthread.h>

#include <string>

#include <stdlib.h>

#include <string>

#include <fstream>

#include <semaphore.h>

using namespace std;

//exection thread structure

struct exec\_thread{

pthread\_t thread;//thread id

struct Request\* req;//present request

sem\_t sem;//semophore

int isFree;//flag

};

//http request structure

struct Request{

//for debuging info and logging

char\* req\_head;

//remote IP address

char\* ip;

//the time the request was reveived by the queueing thread

char\* recv\_time;

//the time the request was scheduled

char\* asgn\_time;

//200,404 or others

int status;

//size of the response is "content-length"

//client socket id

int client\_socket;

//GET or HEAD

int requestType;

const char\* requestDir;

int contentLength;//the length of requested file, 0 for HEAD, -1 for invalid request, others for GET.

char\* contentType;//text/html or image/gif

int isDir;// 1 for directory,0 for file, -1 for invalid URL

};

void\* run(void\* id);

struct Request\* http\_analyzer(char\* p);

void analyseUrl(char\* url,char\* serverDir,struct Request\* r);

//scheduler

void\* FCFS\_scheduler(void\* p);

void\* SJF\_scheduler(void\* p);

char\* Time();

char\* lastmodifyTime(char\* d );

void getResponse(struct Request\* req,string &res);

void logging(struct Request\* req,int debug,char\* logfile);

char\* generateIndex(struct Request\* req);

void Info();