/\* J. David's webserver \*/

/\* This is a simple webserver.

\* Created November 1999 by J. David Blackstone.

\* CSE 4344 (Network concepts), Prof. Zeigler

\* University of Texas at Arlington

\*/

/\* This program compiles for Sparc Solaris 2.6.

\* To compile for Linux:

\* 1) Comment out the #include <pthread.h> line.

\* 2) Comment out the line that defines the variable newthread.

\* 3) Comment out the two lines that run pthread\_create().

\* 4) Uncomment the line that runs accept\_request().

\* 5) Remove -lsocket from the Makefile.

\*/

#include <stdio.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <ctype.h>

#include <strings.h>

#include <string.h>

#include <sys/stat.h>

#include <pthread.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdint.h>

#define ISspace(x) isspace((int)(x))

#define SERVER\_STRING "Server: jdbhttpd/0.1.0\r\n"

#define STDIN 0

#define STDOUT 1

#define STDERR 2

void accept\_request(void \*);

void bad\_request(int);

void cat(int, FILE \*);

void cannot\_execute(int);

void error\_die(const char \*);

void execute\_cgi(int, const char \*, const char \*, const char \*);

int get\_line(int, char \*, int);

void headers(int, const char \*);

void not\_found(int);

void serve\_file(int, const char \*);

int startup(u\_short \*);

void unimplemented(int);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* A request has caused a call to accept() on the server port to

\* return. Process the request appropriately.

\* Parameters: the socket connected to the client \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void accept\_request(void \*arg)

{

int client = (intptr\_t)arg;

char buf[1024];

size\_t numchars;

char method[255];

char url[255];

char path[512];

size\_t i, j;

struct stat st;

int cgi = 0; /\* becomes true if server decides this is a CGI

\* program \*/

char \*query\_string = NULL;

numchars = get\_line(client, buf, sizeof(buf));

i = 0; j = 0;

while (!ISspace(buf[i]) && (i < sizeof(method) - 1))

{

method[i] = buf[i];

i++;

}

j=i;

method[i] = '\0';

if (strcasecmp(method, "GET") && strcasecmp(method, "POST"))

{

unimplemented(client);

return;

}

if (strcasecmp(method, "POST") == 0)

cgi = 1;

i = 0;

while (ISspace(buf[j]) && (j < numchars))

j++;

while (!ISspace(buf[j]) && (i < sizeof(url) - 1) && (j < numchars))

{

url[i] = buf[j];

i++; j++;

}

url[i] = '\0';

if (strcasecmp(method, "GET") == 0)

{

query\_string = url;

while ((\*query\_string != '?') && (\*query\_string != '\0'))

query\_string++;

if (\*query\_string == '?')

{

cgi = 1;

\*query\_string = '\0';

query\_string++;

}

}

sprintf(path, "htdocs%s", url);

if (path[strlen(path) - 1] == '/')

strcat(path, "index.html");

if (stat(path, &st) == -1) {

while ((numchars > 0) && strcmp("\n", buf)) /\* read & discard headers \*/

numchars = get\_line(client, buf, sizeof(buf));

not\_found(client);

}

else

{

if ((st.st\_mode & S\_IFMT) == S\_IFDIR)

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

if ((st.st\_mode & S\_IXUSR) ||

(st.st\_mode & S\_IXGRP) ||

(st.st\_mode & S\_IXOTH) )

cgi = 1;

if (!cgi)

serve\_file(client, path);

else

execute\_cgi(client, path, method, query\_string);

}

close(client);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Inform the client that a request it has made has a problem.

\* Parameters: client socket \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void bad\_request(int client)

{

char buf[1024];

sprintf(buf, "HTTP/1.0 400 BAD REQUEST\r\n");

send(client, buf, sizeof(buf), 0);

sprintf(buf, "Content-type: text/html\r\n");

send(client, buf, sizeof(buf), 0);

sprintf(buf, "\r\n");

send(client, buf, sizeof(buf), 0);

sprintf(buf, "<P>Your browser sent a bad request, ");

send(client, buf, sizeof(buf), 0);

sprintf(buf, "such as a POST without a Content-Length.\r\n");

send(client, buf, sizeof(buf), 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Put the entire contents of a file out on a socket. This function

\* is named after the UNIX "cat" command, because it might have been

\* easier just to do something like pipe, fork, and exec("cat").

\* Parameters: the client socket descriptor

\* FILE pointer for the file to cat \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cat(int client, FILE \*resource)

{

char buf[1024];

fgets(buf, sizeof(buf), resource);

while (!feof(resource))

{

send(client, buf, strlen(buf), 0);

fgets(buf, sizeof(buf), resource);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Inform the client that a CGI script could not be executed.

\* Parameter: the client socket descriptor. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void cannot\_execute(int client)

{

char buf[1024];

sprintf(buf, "HTTP/1.0 500 Internal Server Error\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "Content-type: text/html\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "<P>Error prohibited CGI execution.\r\n");

send(client, buf, strlen(buf), 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Print out an error message with perror() (for system errors; based

\* on value of errno, which indicates system call errors) and exit the

\* program indicating an error. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void error\_die(const char \*sc)

{

perror(sc);

exit(1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Execute a CGI script. Will need to set environment variables as

\* appropriate.

\* Parameters: client socket descriptor

\* path to the CGI script \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void execute\_cgi(int client, const char \*path,

const char \*method, const char \*query\_string)

{

char buf[1024];

int cgi\_output[2];

int cgi\_input[2];

pid\_t pid;

int status;

int i;

char c;

int numchars = 1;

int content\_length = -1;

buf[0] = 'A'; buf[1] = '\0';

if (strcasecmp(method, "GET") == 0)

while ((numchars > 0) && strcmp("\n", buf)) /\* read & discard headers \*/

numchars = get\_line(client, buf, sizeof(buf));

else if (strcasecmp(method, "POST") == 0) /\*POST\*/

{

numchars = get\_line(client, buf, sizeof(buf));

while ((numchars > 0) && strcmp("\n", buf))

{

buf[15] = '\0';

if (strcasecmp(buf, "Content-Length:") == 0)

content\_length = atoi(&(buf[16]));

numchars = get\_line(client, buf, sizeof(buf));

}

if (content\_length == -1) {

bad\_request(client);

return;

}

}

else/\*HEAD or other\*/

{

}

if (pipe(cgi\_output) < 0) {

cannot\_execute(client);

return;

}

if (pipe(cgi\_input) < 0) {

cannot\_execute(client);

return;

}

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

cannot\_execute(client);

return;

}

sprintf(buf, "HTTP/1.0 200 OK\r\n");

send(client, buf, strlen(buf), 0);

if (pid == 0) /\* child: CGI script \*/

{

char meth\_env[255];

char query\_env[255];

char length\_env[255];

dup2(cgi\_output[1], STDOUT);

dup2(cgi\_input[0], STDIN);

close(cgi\_output[0]);

close(cgi\_input[1]);

sprintf(meth\_env, "REQUEST\_METHOD=%s", method);

putenv(meth\_env);

if (strcasecmp(method, "GET") == 0) {

sprintf(query\_env, "QUERY\_STRING=%s", query\_string);

putenv(query\_env);

}

else { /\* POST \*/

sprintf(length\_env, "CONTENT\_LENGTH=%d", content\_length);

putenv(length\_env);

}

execl(path, NULL);

exit(0);

} else { /\* parent \*/

close(cgi\_output[1]);

close(cgi\_input[0]);

if (strcasecmp(method, "POST") == 0)

for (i = 0; i < content\_length; i++) {

recv(client, &c, 1, 0);

write(cgi\_input[1], &c, 1);

}

while (read(cgi\_output[0], &c, 1) > 0)

send(client, &c, 1, 0);

close(cgi\_output[0]);

close(cgi\_input[1]);

waitpid(pid, &status, 0);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Get a line from a socket, whether the line ends in a newline,

\* carriage return, or a CRLF combination. Terminates the string read

\* with a null character. If no newline indicator is found before the

\* end of the buffer, the string is terminated with a null. If any of

\* the above three line terminators is read, the last character of the

\* string will be a linefeed and the string will be terminated with a

\* null character.

\* Parameters: the socket descriptor

\* the buffer to save the data in

\* the size of the buffer

\* Returns: the number of bytes stored (excluding null) \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int get\_line(int sock, char \*buf, int size)

{

int i = 0;

char c = '\0';

int n;

while ((i < size - 1) && (c != '\n'))

{

n = recv(sock, &c, 1, 0);

/\* DEBUG printf("%02X\n", c); \*/

if (n > 0)

{

if (c == '\r')

{

n = recv(sock, &c, 1, MSG\_PEEK);

/\* DEBUG printf("%02X\n", c); \*/

if ((n > 0) && (c == '\n'))

recv(sock, &c, 1, 0);

else

c = '\n';

}

buf[i] = c;

i++;

}

else

c = '\n';

}

buf[i] = '\0';

return(i);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Return the informational HTTP headers about a file. \*/

/\* Parameters: the socket to print the headers on

\* the name of the file \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void headers(int client, const char \*filename)

{

char buf[1024];

(void)filename; /\* could use filename to determine file type \*/

strcpy(buf, "HTTP/1.0 200 OK\r\n");

send(client, buf, strlen(buf), 0);

strcpy(buf, SERVER\_STRING);

send(client, buf, strlen(buf), 0);

sprintf(buf, "Content-Type: text/html\r\n");

send(client, buf, strlen(buf), 0);

strcpy(buf, "\r\n");

send(client, buf, strlen(buf), 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Give a client a 404 not found status message. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void not\_found(int client)

{

char buf[1024];

sprintf(buf, "HTTP/1.0 404 NOT FOUND\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, SERVER\_STRING);

send(client, buf, strlen(buf), 0);

sprintf(buf, "Content-Type: text/html\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "<HTML><TITLE>Not Found</TITLE>\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "<BODY><P>The server could not fulfill\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "your request because the resource specified\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "is unavailable or nonexistent.\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "</BODY></HTML>\r\n");

send(client, buf, strlen(buf), 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Send a regular file to the client. Use headers, and report

\* errors to client if they occur.

\* Parameters: a pointer to a file structure produced from the socket

\* file descriptor

\* the name of the file to serve \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void serve\_file(int client, const char \*filename)

{

FILE \*resource = NULL;

int numchars = 1;

char buf[1024];

buf[0] = 'A'; buf[1] = '\0';

while ((numchars > 0) && strcmp("\n", buf)) /\* read & discard headers \*/

numchars = get\_line(client, buf, sizeof(buf));

resource = fopen(filename, "r");

if (resource == NULL)

not\_found(client);

else

{

headers(client, filename);

cat(client, resource);

}

fclose(resource);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* This function starts the process of listening for web connections

\* on a specified port. If the port is 0, then dynamically allocate a

\* port and modify the original port variable to reflect the actual

\* port.

\* Parameters: pointer to variable containing the port to connect on

\* Returns: the socket \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int startup(u\_short \*port)

{

int httpd = 0;

int on = 1;

struct sockaddr\_in name;

httpd = socket(PF\_INET, SOCK\_STREAM, 0);

if (httpd == -1)

error\_die("socket");

memset(&name, 0, sizeof(name));

name.sin\_family = AF\_INET;

name.sin\_port = htons(\*port);

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

if ((setsockopt(httpd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on))) < 0)

{

error\_die("setsockopt failed");

}

if (bind(httpd, (struct sockaddr \*)&name, sizeof(name)) < 0)

error\_die("bind");

if (\*port == 0) /\* if dynamically allocating a port \*/

{

socklen\_t namelen = sizeof(name);

if (getsockname(httpd, (struct sockaddr \*)&name, &namelen) == -1)

error\_die("getsockname");

\*port = ntohs(name.sin\_port);

}

if (listen(httpd, 5) < 0)

error\_die("listen");

return(httpd);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Inform the client that the requested web method has not been

\* implemented.

\* Parameter: the client socket \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void unimplemented(int client)

{

char buf[1024];

sprintf(buf, "HTTP/1.0 501 Method Not Implemented\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, SERVER\_STRING);

send(client, buf, strlen(buf), 0);

sprintf(buf, "Content-Type: text/html\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "<HTML><HEAD><TITLE>Method Not Implemented\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "</TITLE></HEAD>\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "<BODY><P>HTTP request method not supported.\r\n");

send(client, buf, strlen(buf), 0);

sprintf(buf, "</BODY></HTML>\r\n");

send(client, buf, strlen(buf), 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

int server\_sock = -1;

u\_short port = 4000;

int client\_sock = -1;

struct sockaddr\_in client\_name;

socklen\_t client\_name\_len = sizeof(client\_name);

pthread\_t newthread;

server\_sock = startup(&port);

printf("httpd running on port %d\n", port);

while (1)

{

client\_sock = accept(server\_sock,

(struct sockaddr \*)&client\_name,

&client\_name\_len);

if (client\_sock == -1)

error\_die("accept");

/\* accept\_request(&client\_sock); \*/

if (pthread\_create(&newthread , NULL, (void \*)accept\_request, (void \*)(intptr\_t)client\_sock) != 0)

perror("pthread\_create");

}

close(server\_sock);

return(0);

}

Client

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

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

{

int sockfd;

int len;

struct sockaddr\_in address;

int result;

char ch = 'A';

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

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

address.sin\_port = htons(9734);

len = sizeof(address);

result = connect(sockfd, (struct sockaddr \*)&address, len);

if (result == -1)

{

perror("oops: client1");

exit(1);

}

write(sockfd, &ch, 1);

read(sockfd, &ch, 1);

printf("char from server = %c\n", ch);

close(sockfd);

exit(0);

}

A mirror for tinyhttpd(Tinyhttpd非官方镜像,Fork自[sourceForge](https://sourceforge.net/projects/tiny-httpd/),仅供学习)

测试CGI时需要本机安装PERL，同时安装perl-cgi

### **Prepare**

Compile for Linux

To compile for Linux:

1) Comment out the #include <pthread.h> line.

2) Comment out the line that defines the variable newthread.

3) Comment out the two lines that run pthread\_create().

4) Uncomment the line that runs accept\_request().

5) Remove -lsocket from the Makefile.

     每个函数的作用：

     accept\_request:  处理从套接字上监听到的一个 HTTP 请求，在这里可以很大一部分地体现服务器处理请求流程。

     bad\_request: 返回给客户端这是个错误请求，HTTP 状态吗 400 BAD REQUEST.

     cat: 读取服务器上某个文件写到 socket 套接字。

     cannot\_execute: 主要处理发生在执行 cgi 程序时出现的错误。

     error\_die: 把错误信息写到 perror 并退出。

     execute\_cgi: 运行 cgi 程序的处理，也是个主要函数。

     get\_line: 读取套接字的一行，把回车换行等情况都统一为换行符结束。

     headers: 把 HTTP 响应的头部写到套接字。

     not\_found: 主要处理找不到请求的文件时的情况。

     sever\_file: 调用 cat 把服务器文件返回给浏览器。

     startup: 初始化 httpd 服务，包括建立套接字，绑定端口，进行监听等。

     unimplemented: 返回给浏览器表明收到的 HTTP 请求所用的 method 不被支持。

     建议源码阅读顺序： main -> startup -> accept\_request -> execute\_cgi, 通晓主要工作流程后再仔细把每个函数的源码看一看。

#### **工作流程**

     （1） 服务器启动，在指定端口或随机选取端口绑定 httpd 服务。

     （2）收到一个 HTTP 请求时（其实就是 listen 的端口 accpet 的时候），派生一个线程运行 accept\_request 函数。

     （3）取出 HTTP 请求中的 method (GET 或 POST) 和 url,。对于 GET 方法，如果有携带参数，则 query\_string 指针指向 url 中 ？ 后面的 GET 参数。

     （4） 格式化 url 到 path 数组，表示浏览器请求的服务器文件路径，在 tinyhttpd 中服务器文件是在 htdocs 文件夹下。当 url 以 / 结尾，或 url 是个目录，则默认在 path 中加上 index.html，表示访问主页。

     （5）如果文件路径合法，对于无参数的 GET 请求，直接输出服务器文件到浏览器，即用 HTTP 格式写到套接字上，跳到（10）。其他情况（带参数 GET，POST 方式，url 为可执行文件），则调用 excute\_cgi 函数执行 cgi 脚本。

    （6）读取整个 HTTP 请求并丢弃，如果是 POST 则找出 Content-Length. 把 HTTP 200  状态码写到套接字。

    （7） 建立两个管道，cgi\_input 和 cgi\_output, 并 fork 一个进程。

    （8） 在子进程中，把 STDOUT 重定向到 cgi\_outputt 的写入端，把 STDIN 重定向到 cgi\_input 的读取端，关闭 cgi\_input 的写入端 和 cgi\_output 的读取端，设置 request\_method 的环境变量，GET 的话设置 query\_string 的环境变量，POST 的话设置 content\_length 的环境变量，这些环境变量都是为了给 cgi 脚本调用，接着用 execl 运行 cgi 程序。

    （9） 在父进程中，关闭 cgi\_input 的读取端 和 cgi\_output 的写入端，如果 POST 的话，把 POST 数据写入 cgi\_input，已被重定向到 STDIN，读取 cgi\_output 的管道输出到客户端，该管道输入是 STDOUT。接着关闭所有管道，等待子进程结束。这一部分比较乱，见下图说明：

