HTTPServer.c

```c
#include <stdio.h>      // I/O
#include <stdlib.h>     // C
#include <string.h>     // String processing
#include <signal.h>     // Signal: sigaction
#include <unistd.h>     // POSIX API, file, I/O, etc.
#include <errno.h>      // perror()
#include <pthread.h>    // pthread()
#include <sys/stat.h>   // stat()
#include <sys/types.h>  // fd_set, etc.
#include <sys/socket.h> // Socket: socket(), bind(), etc.
#include <arpa/inet.h>  // inet_ntoa(), etc.
#include <netinet/in.h> // struct sockaddr_in
#include <libgen.h>     // For basename() and dirname().

#include "ContentType.h"

#include "HTTPServer.h"

void PANIC(char *msg);
#define PANIC(msg){perror(msg);exit(-1);}


#define DEFAULT_PORT 80
#define PATH_SIZE 512
#define RECEIVE_BUFFER_SIZE 2048
#define SEND_BUFFER_SIZE 2048
#define CONTENT_BUFFER_SIZE 1024

#define METHOD_GET 1
#define BAD_REQUEST -1


typedef struct http_request {
   char method[16];        // Request Method, Ex: get...
   char path[PATH_SIZE];
   char prefix[16];

   int rangeflag;          // Flag
   long rangestart;        // The starting position of the data
```

```
    long rangeend;          // The end of the data
    long rangetotal;        // The total length of the data

    int responsecode;       // State code
}HR;


void* threadFunc(void *threadArgs);
int parseRequest(char *recvBuffer, struct http_request *httpRequest);
void methodGET(int clientfd, struct http_request *httpRequest);
void response(int clientfd, struct http_request *httpRequest);
int transferFile(int clientfd, FILE *fp,
            int type, int rangestart, int totallength);
int sendData(int clientfd, char *buf, int length);


int main(int argc, char *argv[]) {
    int port = DEFAULT_PORT;            // Server port
    int sockfd;                         // Server socket()
    int clientfd;
    const int yes = 1;                  // For setsockopt()
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    int len;                            // sizeof(client_addr)

    pthread_t threadid;

    // SIGPIPE
    struct sigaction action;
    action.sa_handler = SIG_IGN;
    sigaction(SIGPIPE, &action, 0);

    // Get port nums from args
    if(argc > 1){
        int argsBuff = atoi(argv[1]);
        if(argsBuff <= 0 || argsBuff > 65535) perror("Args");
        else port = argsBuff;
    }

    // Create a socket.
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
if(sockfd < 0) PANIC("Socket()");

// Release the port which used before.
if(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0) {
   PANIC("setsockopt()");
}

server_addr.sin_family = AF_INET;            // IPv4
server_addr.sin_port = htons(port);          // Set port
server_addr.sin_addr.s_addr = INADDR_ANY;    // 0.0.0.0

// Bind.
if(bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) != 0)
   PANIC("Bind");

// Listen sockfd. Up to 20 connections.
if(listen(sockfd, 20) != 0)
   PANIC("Listen");

// Server Loop.
while(1){
   // Configure a memory to place new_fd.
   int *threadArgs = calloc(1, sizeof(int));

   len = sizeof(client_addr);

   // Wait for connection from client...

   // Accept.
   clientfd = accept(sockfd, (struct sockaddr*) &client_addr,
               (socklen_t*) &len);
   if(clientfd < 0) PANIC("Accept");

   printf("Client from %s : %d\n",
        inet_ntoa(client_addr.sin_addr),
        ntohs(client_addr.sin_port));

   *threadArgs = clientfd;

   // Create a thread.
   if(pthread_create(&threadid, NULL, threadFunc, threadArgs) != 0){
```

```
            PANIC("pthread_create()");
        } else{
            // After finishing the thread function,
            // the resource will automatically freed.
            pthread_detach(threadid);
        }

    } // while(1), Server Loop.

    return 0;
} // main()


void* threadFunc(void *threadArgs){
    int clientfd = *(int*) threadArgs;
    free(threadArgs);

    char recvBuffer[RECEIVE_BUFFER_SIZE];
    int recvSize = 0;
    int recvPt = 0;


    struct http_request httpRequest;
    httpRequest.rangeflag = 0;
    httpRequest.rangestart = 0;

    // Receive loop.
    while(1){
        recvSize = recv(clientfd, recvBuffer + recvPt,
        RECEIVE_BUFFER_SIZE - recvPt - 1, 0);
        if(recvSize <= 0){
            close(clientfd);
            pthread_exit(NULL);
            PANIC("recv()");
        }
        recvPt += recvSize;
        recvBuffer[recvPt] = '\0';

        // When received "\r\n\r\n" or "\n\n" , break.
        if(strstr(recvBuffer, "\r\n\r\n") != NULL ||
          strstr(recvBuffer, "\n\n") != NULL)
```

```
            break;
        } // while(1) Receive loop.

        printf("Request: \n%s\n", recvBuffer);

        // Parse request.
        switch(parseRequest(recvBuffer, &httpRequest)){
            case METHOD_GET:
                methodGET(clientfd, &httpRequest);
                break;
            case BAD_REQUEST:
                break;
            default:
                break;
        }

        close(clientfd);
        return 0;
    } // threadFunc()


    int parseRequest(char *recvBuffer, struct http_request *httpRequest){
        char path[PATH_SIZE];
        char protocol[20];

        // Parse Http Header.
        if(sscanf(recvBuffer, "%s %s %s",
                httpRequest -> method,
                httpRequest -> path, protocol) != 3)
            return BAD_REQUEST;

        strcpy(path, httpRequest -> path);

        if(path[strlen(path)-1] == '/')
            strcat(path, "INDEX.html");

        strcpy(httpRequest -> path, path);

        char *base = basename(path);
        char *ext = strrchr(base,'.');
        if(!ext) strcpy(httpRequest -> prefix, "*");
```

```
    else {
        ext = ext + 1;
        strcpy(httpRequest -> prefix, ext);
    }

    printf("Path: %s\nPrefix: %s\n",
        httpRequest -> path,
        httpRequest -> prefix);

    // Method "GET"
    if(strcmp(httpRequest->method, "GET") == 0)
        return METHOD_GET;

    return -1;
} // parseRequest()


void methodGET(int clientfd, struct http_request *httpRequest){
    struct stat s;
    char path[PATH_SIZE];
    char resourcePath[PATH_SIZE];
    strcpy(resourcePath, "./www/");
    sprintf(path, "%s%s", resourcePath, httpRequest -> path + 1);
    printf("fopen() Path: %s\n", path);

    FILE *fp = fopen(path, "r");

    // file exists or not.
    if(fp == NULL){
        printf("File not exist: %s\n", path);
        httpRequest -> responsecode = 404;
        response(clientfd, httpRequest);
    } else {
        printf("File exist: %s\n", path); // puts("file exists");

        if(httpRequest -> rangeflag == 0){
            stat(path, &s);
            httpRequest -> rangetotal = s.st_size;
            printf("total length: %ld\n\n\n", httpRequest -> rangetotal);
        }
        httpRequest -> responsecode = 200;
```

```
        response(clientfd, httpRequest);

        transferFile(clientfd, fp, httpRequest -> rangeflag,
                httpRequest -> rangestart, httpRequest -> rangetotal);

        fclose(fp);
    }

} // methodGET()


void response(int clientfd, struct http_request *httpRequest){
    char sendBuffer[SEND_BUFFER_SIZE];
    char content[CONTENT_BUFFER_SIZE];

    switch(httpRequest -> responsecode){
        case 200:
            sprintf(sendBuffer,
                "HTTP/1.1 200 OK\r\n"
                "Server: WenYuan/1.0\r\n"
                "Content-Type: %s\r\n"
                "Accept-Ranges: bytes\r\n"
                "Content-Length: %ld\r\n"
                "Connection: close\r\n"
                "\r\n",
                getContentType(httpRequest -> prefix),
                httpRequest -> rangetotal);
            break;

        case 404:
            strcpy(content,
                "<!DOCTYPE html>"
                "<html><head><title>404 Not Found</title></head>"
                "<body><h1>404 Not Found</h1>"
                "File Not Found.</body></html>");
            sprintf(sendBuffer,
                "HTTP/1.1 404 Object Not Found\r\n"
                "Server: WenYuan/1.0\r\n"
                "Content-Type: %s\r\n"
                "Content-Length: %ld\r\n"
                "Connection: close\r\n"
```

```
            "\r\n"
            "%s",
            getContentType("html"), strlen(content), content);
      break;

    default:
      break;
  } // switch(code)

  // Send the response to client.
  sendData(clientfd, sendBuffer, strlen(sendBuffer));
} // response()


int transferFile(int clientfd, FILE *fp,
          int type, int rangestart, int totallength){
  if(type == 1) fseek(fp, rangestart, 0);

  int sendnum = 0;
  int segment = 1024;

  /*
   * feof(fp)-->If the data has not been read completely,
   * it will return zero.
   *
   * To read the data from the file fp.
   */
  while(!feof(fp) && sendnum < totallength){
    char buf[segment];
    memset(buf, 0, 1024);
    int i = 0;

    while(!feof(fp) && i < segment && sendnum+i < totallength){
      buf[i++] = fgetc(fp);
    }

    //  Send data every 1024 bytes
    if(sendData(clientfd, buf, i) == 0)
      return 0;

    sendnum += i;
```

```
    }

    return 1;
} // transferFile()


// Used to transfer file.
int sendData(int clientfd, char *buf, int length){
    if(length <= 0)
        return 0;

    int result = send(clientfd, buf, length, 0);
    if(result < 0)
        return 0;

    return 1;
} // sendData()
```