```c
/*
 * proxy.c - ICS Web proxy
 * WEI Xiao-Miao
 * 516015910018
 */

#include "csapp.h"
#include <stdarg.h>
#include <sys/select.h>

/*
 * Function prototypes
 */
void *thread_routine(void *vargp);
void doit(int fd, struct sockaddr_storage clientaddr);

int parse_uri(char *uri, char *target_addr, char *path, char *port);
void format_log_entry(char *logstring, struct sockaddr_in *sockaddr, char *uri, size_t
size);

int read_requesthdrs(rio_t *rio_client, int serverfd);
int read_requestbody(rio_t *rio_client, int serverfd, int req_body_len);
int read_responsehdrs(rio_t* rio_server, int fd, int serverfd, size_t *res_size);
int read_responsebody(rio_t *rio_server, int fd ,int serverfd, int res_body_len, size_t
*res_size);

ssize_t Rio_readn_w(int fd, void *usrbuf, size_t n);
ssize_t Rio_readnb_w(rio_t *rp, void *usrbuf, size_t n);
ssize_t Rio_readlineb_w(rio_t *rp, void *usrbuf, size_t n);
ssize_t Rio_writen_w(int fd, void *usrbuf, size_t n);

void syn_print(char *msg, ...);

/*
 * Parameters struct for thread routine
 * follow CS:APP 12.3.2
 */
struct thread_routine_params
{
    int* connfd_p;
    struct sockaddr_storage* clientaddr_p;
};
sem_t s;

/*
 * main - Main routine for the proxy program
 */
int main(int argc, char **argv) {

    /* avoid of termination cauesd by SIGPIPE */
```

```
    Signal(SIGPIPE, SIG_IGN);
    /* make printf synchronize */
    Sem_init(&s, 0, 1);

    /* Check arguments */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
        exit(0);
    }

    /* listening file descriptor to client */
    int listenfd = Open_listenfd(argv[1]);

    while(1) {

        /* init thread_routine_params */
        struct thread_routine_params* params_p = Malloc(sizeof(struct
thread_routine_params));
        socklen_t clientlen = sizeof(struct sockaddr_storage);
        params_p->connfd_p = Malloc(sizeof(int));
        params_p->clientaddr_p = Malloc(clientlen);

        /* set params */
        struct sockaddr_storage clientaddr;
        int connfd = Accept(listenfd, (SA *)(&clientaddr), &clientlen);
        *(params_p->connfd_p) = connfd;
        *(params_p->clientaddr_p) = clientaddr;

        /* create a thread */
        pthread_t tid;
        Pthread_create(&tid, NULL, thread_routine, params_p);

        /* no waiting for thread, handle next connect concurrently */
    }

    exit(0);
}

/*
 * thread_routine - Deal with a connection in a thread
 */
void* thread_routine(void* vargp) {

    /* get params */
    Pthread_detach(Pthread_self());
    struct thread_routine_params* params_p = (struct thread_routine_params*) vargp;
    int connfd = *(params_p->connfd_p);
    struct sockaddr_storage clientaddr = *(params_p->clientaddr_p);

    /* do real business */
    doit(connfd, clientaddr);

    /* terminate a transaction */
```

```
    Close(connfd);
    Free(params_p->clientaddr_p);
    Free(params_p->connfd_p);
    Free(params_p);
    return NULL;

}

/*
 * doit - Forwards req and res between client and server
 * Mainly, doit is made up of 5 steps:
 *      1. Connect to server and forwards req_line from client
 *      2. Read req headers from client and forwards to server
 *      3. Read req body content from client and forwards to server
 *      4. Read res headers from server and forwards to client
 *      5. Read res body content from server and forwards to client
 */
void doit(int fd, struct sockaddr_storage clientaddr) {


    rio_t rio_client, rio_server;
    Rio_readinitb(&rio_client, fd);

    /* 0. Get and parse req_line from client */
    char buf[MAXLINE];
    if (Rio_readlineb_w(&rio_client, buf, MAXLINE) == 0) {
        syn_print("No request line.\n");
        return;
    }
    char method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    if (sscanf(buf, "%s %s %s", method, uri, version) != 3) {
        syn_print("Parsing request line error.\n");
        return;
    }
    char hostname[MAXLINE], pathname[MAXLINE], port[MAXLINE];
    if (parse_uri(uri, hostname, pathname, port) != 0) {
        syn_print("Parsing uri error.\n");
        return;
    }

    /* 1. Connect to server and forwards req_line form client */
    int serverfd = open_clientfd(hostname, port);
    if (serverfd < 0) {
        syn_print("open_clientfd error.\n");
        return;
    }
    Rio_readinitb(&rio_server, serverfd);
    char req_line[4*MAXLINE];
    sprintf(req_line, "%s /%s %s\r\n", method, pathname, version);
    if (strlen(req_line) != Rio_writen_w(serverfd, req_line, strlen(req_line))) {
        syn_print("Error constructing a request line to server.\n");
        Close(serverfd);
        return;
```

```
    }

    /* 2. Read req headers from client and forwards to server */
    int req_body_len = read_requesthdrs(&rio_client, serverfd);
    if (req_body_len == -1) return;

    /* 3. Read req body content from client and forwards to server */
    if (req_body_len > 0)
    {
        int len = read_requestbody(&rio_client, serverfd, req_body_len);
        if (len < 0) return;
    }

    /* 4. Read res headers from server and forwards to client */
    size_t res_size = 0;
    int res_body_len = read_responsehdrs( &rio_server, fd, serverfd, &res_size);
    if (res_body_len == -1) return;

    /* 5. Read res body content from server and forwards to client */
    if (res_body_len > 0)
    {
        int len = read_responsebody(&rio_server, fd ,serverfd, res_body_len, &res_size);
        if (len < 0) return;
    }

    /* 6. Fill log */
    char logstring[MAXLINE];
    format_log_entry(logstring, (struct sockaddr_in *)&clientaddr, uri, res_size);

    syn_print("%s\n", logstring);
    Close(serverfd);
    return;
}


/*
 * parse_uri - URI parser
 *
 * Given a URI from an HTTP proxy GET request (i.e., a URL), extract
 * the host name, path name, and port.  The memory for hostname and
 * pathname must already be allocated and should be at least MAXLINE
 * bytes. Return -1 if there are any problems.
 */
int parse_uri(char *uri, char *hostname, char *pathname, char *port)
{
    char *hostbegin;
    char *hostend;
    char *pathbegin;
    int len;

    if (strncasecmp(uri, "http://", 7) != 0) {
        hostname[0] = '\0';
        return -1;
```

```
    }

    /* Extract the host name */
    hostbegin = uri + 7;
    hostend = strpbrk(hostbegin, " :/\r\n\0");
    if (hostend == NULL)
        return -1;
    len = hostend - hostbegin;
    strncpy(hostname, hostbegin, len);
    hostname[len] = '\0';

    /* Extract the port number */
    if (*hostend == ':') {
        char *p = hostend + 1;
        while (isdigit(*p))
            *port++ = *p++;
        *port = '\0';
    } else {
        strcpy(port, "80");
    }

    /* Extract the path */
    pathbegin = strchr(hostbegin, '/');
    if (pathbegin == NULL) {
        pathname[0] = '\0';
    }
    else {
        pathbegin++;
        strcpy(pathname, pathbegin);
    }

    return 0;
}

/*
 * format_log_entry - Create a formatted log entry in logstring.
 *
 * The inputs are the socket address of the requesting client
 * (sockaddr), the URI from the request (uri), the number of bytes
 * from the server (size).
 */
void format_log_entry(char *logstring, struct sockaddr_in *sockaddr,
                      char *uri, size_t size)
{
    time_t now;
    char time_str[MAXLINE];
    unsigned long host;
    unsigned char a, b, c, d;

    /* Get a formatted time string */
    now = time(NULL);
    strftime(time_str, MAXLINE, "%a %d %b %Y %H:%M:%S %Z", localtime(&now));
```

```
    /*
     * Convert the IP address in network byte order to dotted decimal
     * form. Note that we could have used inet_ntoa, but chose not to
     * because inet_ntoa is a Class 3 thread unsafe function that
     * returns a pointer to a static variable (Ch 12, CS:APP).
     */
    host = ntohl(sockaddr->sin_addr.s_addr);
    a = host >> 24;
    b = (host >> 16) & 0xff;
    c = (host >> 8) & 0xff;
    d = host & 0xff;

    /* Return the formatted log entry string */
    sprintf(logstring, "%s: %d.%d.%d.%d %s %zu", time_str, a, b, c, d, uri, size);
}


/*
 * read_requesthdrs - Handle requset headers
 *
 * Prototype is from CS:APP Chp11.6.4. Expand the prototype to
 * forwards those headers to server and return Content-Length
 */
int read_requesthdrs(rio_t *rio_client, int serverfd) {
    size_t n;
    char buf[MAXLINE];
    int req_body_len = 0;
    while((n = Rio_readlineb_w(rio_client, buf, MAXLINE)) != 0) {
        /* Forwarding */
        if (Rio_writen_w(serverfd, buf, n) != n) {
            syn_print("Forwarding error.\n");
            Close(serverfd);
            return -1;
        }
        /* get Content-Length out */
        if (strncasecmp(buf, "Content-Length:", strlen("Content-Length:")) == 0) {
            sscanf(buf+strlen("Content-Length:"), "%d", &req_body_len);
        }
        /* end of header */
        if (strcmp("\r\n", buf) == 0)  {
            break;
        }
    }
    return req_body_len;
}

/*
 * read_requestbody - Handle requset content
 *
 * Forwards body content to server
 */
int read_requestbody(rio_t *rio_client, int serverfd, int req_body_len) {
    char *buf[MAXLINE];
```

```
    int rest_len = req_body_len;
    while(rest_len > 0) {
        int max_len ;
        if (rest_len <= MAXLINE) {
            max_len = rest_len;
        } else {
            max_len = MAXLINE;
        }
        /* get req content line */
        int read_len = Rio_readnb_w(rio_client, buf, max_len);
        if (read_len == 0) {
            syn_print("Error getting body content.\n");
            Close(serverfd);
            syn_print("serverfd has been closed.\n");
            return -1;
        } else {
            /* forward req content line */
            if (Rio_writen_w(serverfd, buf, read_len) == 0) {
                syn_print("Error forwarding body content to the server.\n");
                Close(serverfd);
                return -1;
            }
        }
        rest_len -= read_len;
    }
    return req_body_len;
}

/*
 * read_responsehdrs - Handle response headers
 *
 * Forwards headers to client
 */
int read_responsehdrs(rio_t* rio_server, int fd, int serverfd, size_t *res_size) {
    char buf[MAXLINE];
    size_t n;
    int res_body_len = 0;
    while((n = Rio_readlineb_w(rio_server, buf, MAXLINE)) != 0) {
        /* receive content line */
        if (Rio_writen_w(fd, buf, n) == 0) {
            syn_print("Error writing to the client.\n");
            Close(serverfd);
            return -1;
        }
        /* response size increase */
        (*res_size) += n;
        /* get out response content length */
        if (strncasecmp(buf, "Content-Length:", strlen("Content-Length:")) == 0) {
            sscanf(buf+16, "%d", &res_body_len);
        }
        if (strcmp("\r\n", buf) == 0) {
            break;
        }
```

```
    }
    return res_body_len;
}

/*
 * read_responsebody - Handle response body
 *
 * Forwards body content to client
 */
int read_responsebody(rio_t *rio_server, int fd ,int serverfd, int res_body_len, size_t
*res_size) {
    char buf[MAXLINE];
    for (int i = 0;i < res_body_len;++i) {
        /* get response */
        if (Rio_readnb_w(rio_server, buf, 1) == 0) {
            syn_print("Error reading response body from server.\n");
            Close(serverfd);
            return -1;
        } else {
            /* forwards response */
            if (Rio_writen_w(fd, buf, 1) == 0) {
                syn_print("Error writing response body to client.\n");
                Close(serverfd);
                return -1;
            }
        }
        (*res_size)++;
    }
    return 0;
}



/*
 * Rio_xxxx_w - wrapped RIO interfaces.
 *
 * Rio_xxxx in csapp.c terminate the process when getting an error.
 * Rio_xxxx_w check and return -1 when getting an error.
 */
ssize_t Rio_readn_w(int fd, void *usrbuf, size_t n) {
    size_t got = rio_readn(fd, usrbuf, n);
    if (got < 0 ) {
        syn_print("Rio_readn error.\n");
        return -1;
    }
    return got;
}
ssize_t Rio_readnb_w(rio_t *rp, void *usrbuf, size_t n) {
    ssize_t rc = rio_readnb(rp, usrbuf, n);
    if (rc < 0) {
        syn_print("Rio_readnb error: %s\n", strerror(errno));
        return -1;
    }
```

```
    return rc;
}
ssize_t Rio_readlineb_w(rio_t *rp, void *usrbuf, size_t n) {
    ssize_t rc = rio_readlineb(rp, usrbuf, n);
    if (rc < 0) {
        syn_print("Rio_readlineb error: %s\n", strerror(errno));
        return -1;
    }
    return rc;
}
ssize_t Rio_writen_w(int fd, void *usrbuf, size_t n) {
    ssize_t wc = rio_writen(fd, usrbuf, n);
    if (wc != n) {
        syn_print("Rio_writen error.\n");
        return -1;
    }
    return wc;
}

/*
 * syn_print - Synchronized printf
 *
 * make printf synchronize
 */
void syn_print(char *msg, ...) {
    if (strstr(msg, "%") == NULL) {
        P(&s);
        printf(msg);
        V(&s);
    } else {
        va_list args;
        va_start(args, msg);
        char* arg =va_arg(args, char*);
        P(&s);
        printf(msg, arg);
        V(&s);
        va_end(args);
    }
}
```