
Programmieren für das Internet

`http://www-pu.informatik.uni-tuebingen.de/pfi-0506`

Übungsblatt 2

Abgabe: 15.11.2005 bis 15:00 Uhr

1. [5 Punkte] Programmiere eine Funktion, die eine als String gegebene HTTP-Request-Line in die Bestandteile Methode, URI und Protokoll-Version zerlegt. Eine URI ist dabei immer in diesem vereinfachten Format gegeben:

```
RequestURI = "/" path-segment*
path-segment = pchar* ( "/" path-segment )*
```

Beachte, dass `pchar` hexadezimal kodierte Zeichen enthalten kann, die dekodiert werden müssen. Ein Beispiel: `datei%201` wird zu `datei_1` dekodiert. Ein Beispiel für HTTP-Request-Line wäre:

```
GET /pub/WWW/TheProject.html HTTP/1.1
```

Lösung: Siehe `decode_uri` Implementierung in der kompletten Lösung.

2. [14 Punkte] Programmiere einen einfachen Webserver für HTTP 1.1, der die `GET`-Methode für die Request-URI aus Aufgabe 1 unterstützt. Der Server interpretiert die URI als Pfad in seinem Dateisystem und sendet den Inhalt der Datei als HTTP-Nachricht mit dem Content-Type `text/plain` zurück. In der Anfrage enthaltene HTTP-Header liest der Server und gibt sie auf der Standardausgabe aus; die Header der Anfrage werden also nicht weiterverarbeitet.

Der Server sucht die Datei unterhalb eines sogenannten *Document-Roots*: Der Pfad `/` in der Request-URI entspricht dem Pfad `/home/knauel` (bzw. deinem Home-Verzeichnis) im Dateisystem. Fordert der Client zum Beispiel die URL `http://localhost/etc/moin.txt` an, so überträgt der Server den Inhalt der Datei `/home/knauel/etc/moin.txt`.

Als Statuscodes sollte der Server 200, 400, 403, 404, 501 und 505 unterstützen.

Lösung: Hier eine komplette Lösung in ANSI C:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 1024
#define DOCROOT "/afs/informatik.uni-tuebingen.de/home/singh/tmp"
#define WRITE 1
#define READ 0
#define HTTP_STATUS_200 "HTTP/1.1 200 OK\r\n"
#define HTTP_STATUS_301 "HTTP/1.1 301 Moved permanently\r\n"
#define HTTP_STATUS_400 "HTTP/1.1 400 Bad syntax\r\n"
#define HTTP_STATUS_403 "HTTP/1.1 403 Forbidden\r\n"
#define HTTP_STATUS_404 "HTTP/1.1 404 File not found\r\n"
#define HTTP_STATUS_501 "HTTP/1.1 501 Internal server error\r\n"
#define HTTP_STATUS_505 "HTTP/1.1 505 Method not implemented\r\n"
#define HTTP_CONTENT_TYPE "Content-Type: text/plain\r\n"
#define HTTP_NEWLINE "\r\n"
#define HTTP_METHOD_GET "GET"
#define HTTP_METHOD_HEAD "HEAD"
#define REDIRECT_PATH_SRC "/geheim/"
#define REDIRECT_PATH_DST "/topsecret/"
#define REDIRECT_LOCATION "Location: "
#define TEXT_NOTFOUND "File not found"
#define TEXT_FORBIDDEN "Forbidden"
#define FIND_ABS "/usr/bin/file"
#define FIND "file"
#define FIND_OPTIONS "-i"
#define DU_ABS "/usr/bin/du"
#define DU "du"
#define DU_OPTIONS NULL
#define LAST_CHUNK "0\r\n"
#define CHUNKED_ENCODING "Transfer-Encoding: chunked\r\n"
#define WRITE_OR_CONT(sd, str, strlen) { if (write(sd, str, strlen) < 0) continue; }

int panic(char *where){
    fprintf(stderr, "error at %s '%s'\n", where, strerror(errno));
    exit(127);
}

int decode_uri(uri, resbuf)

```

```

        char *uri, *resbuf;
    {
        int i = 0, j = 0;
        char c, code[3] = {(char) NULL, (char) NULL, (char) NULL};

        bzero((void *) resbuf, strlen(resbuf));

        while ((c = uri[i++]) != (char) NULL) {
            if (c == '%')
                if (isxdigit(uri[i]) && (uri[i+1] != (char) NULL) && isxdigit(uri[i+1])) {
                    strncpy(code, &uri[i], 2);
                    resbuf[j++] = (char) strtol(code, NULL, 16);
                    i += 2;
                }
            else return -1;
            else
                resbuf[j++] = c;
        }
        return 0;
    }

int readline(int sd, char *buf, int buflen)
{
    int n, rc;
    char c;

    for (n = 1; n < buflen; n++) {
        if ((rc = read(sd, &c, 1)) == 1) {
            *buf++ = c;
            if (c == '\n')
                break;
        }
        else if (rc == 0) {
            if (n == 1)
                return 0;
            else
                break;
        }
        else
            return -1;
    }
    *buf = 0;
    return n;
}

int request_line(line, method, uri, protver)
    char *line, *method, *uri, *protver;
{
    char *t;

```

```

if ((t = strtok(line, " ")) != NULL)
    strcpy(method, t);
else
    return -1;
if ((t = strtok(NULL, " ")) != NULL)
    strcpy(uri, t);
else
    return -1;
if ((t = strtok(NULL, " ")) != NULL)
    strcpy(protver, t);
else
    return -1;
return 0;
}

int exec_cmd(f_name, abs_cmd, cmd, options, res)
    char* f_name, *abs_cmd, *cmd, *options, *res;
{

    int child_pid, pfd[2];

    bzero((void *) res, strlen(res));

    if( pipe(pfd) < 0)
        panic("pipe");

    if((child_pid = fork()) < 0){
        panic("fork");
    }else if (child_pid == 0){

        /*child process*/
        close(pfd[READ]);
        /* stdout == write end of the pipe */
        if(dup2(pfd[WRITE], WRITE) == -1 )
            panic( "dup2 failed" );
        close (pfd[WRITE]);
        if (options != NULL)
            execl(abs_cmd, cmd, options, f_name,0);
        else
            execl(abs_cmd, cmd, f_name,0);
        exit(0);

    } else {

/*parent process*/
        close(pfd[WRITE]); /* first close the write end of the pipe */
        /* stdin == read end of the pipe */

```

```

        if(dup2(pfd[READ], READ) == -1)
            panic("dup2 failed");
        close(pfd[READ]);
        readline((int) stdin, res, BUFLLEN);
    }

    return 0;
}

int main(int argc, char *argv[])
{
    int sd, addrlen, csd = -1, port;
    char chunk_size[BUFLLEN];
    char tmp[BUFLLEN];
    struct sockaddr_in client_addr, local_addr;

    if (argc != 2){
        printf("USAGE: %s <portnumber>\n", argv[0]);
        exit(-1);
    }
    port = atoi(argv[1]);

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        panic("socket");

    bzero((void *) &local_addr, sizeof(struct sockaddr_in));
    local_addr.sin_family = AF_INET;
    local_addr.sin_port = htons(port);
    local_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(sd, (struct sockaddr *) &local_addr,
             sizeof(struct sockaddr_in)) < 0)
        panic("bind");

    if (listen(sd, SOMAXCONN) < 0)
        panic("listen");

    while (1) {
        char request[BUFLLEN], bogus[BUFLLEN], method[BUFLLEN];
        char uri[BUFLLEN], protver[BUFLLEN], filename[2*BUFLLEN];
        int file, read_status;

        addrlen = sizeof(struct sockaddr);

        if (csd > 0)
            close(csd);
        printf("Waiting for connection\n");
        csd = accept(sd, (struct sockaddr*) &client_addr, &addrlen);

```

```

/* read the request line */
if (readline(csd, (char *) &request, BUFLLEN) < 0)
    continue;

/* read and ignore headers. */
/* A real HTTP implementation must take care of message body */
do {
    if ((read_status = readline(csd, (char *) &bogus, BUFLLEN)) < 0)
break;
    printf("%s", bogus);
} while (index(bogus, '\r') != bogus);

if (read_status < 0)
    continue;
if (request_line(request, method, uri, protver) < 0) {
    write(csd, HTTP_STATUS_400, strlen(HTTP_STATUS_400));
    continue;
} else if (!(strcasecmp(method, HTTP_METHOD_GET) == 0)
|| (strcasecmp(method, HTTP_METHOD_HEAD) == 0)) {
    write(csd, HTTP_STATUS_505, strlen(HTTP_STATUS_505));
    continue;
}

strcpy(tmp,uri);
decode_uri(tmp,uri);
strcpy(filename, DOCROOT);
strcat(filename, uri);

printf("Client requested: %s\n", filename);

if (strncmp(uri, REDIRECT_PATH_SRC,
strlen(REDIRECT_PATH_SRC)) == 0) {

    char dst[BUFLLEN];

    WRITE_OR_CONT(csd, HTTP_STATUS_301, strlen(HTTP_STATUS_301));
    strcpy(dst, REDIRECT_LOCATION);
    strcat(dst, REDIRECT_PATH_DST);
    strcat(dst, &uri[strlen(REDIRECT_PATH_SRC)]);
    WRITE_OR_CONT(csd, dst, strlen(dst));
    write(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));
    continue;
}

file = open(filename, O_RDONLY);
if (file < 0) {

    switch (errno) {

```

```

        case EACCES:
WRITE_OR_CONT(csd, HTTP_STATUS_403, strlen(HTTP_STATUS_403));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));
WRITE_OR_CONT(csd, TEXT_FORBIDDEN, strlen(TEXT_FORBIDDEN));
break;

        case ENOENT:
WRITE_OR_CONT(csd, HTTP_STATUS_404, strlen(HTTP_STATUS_404));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));
WRITE_OR_CONT(csd, TEXT_NOTFOUND, strlen(TEXT_NOTFOUND));
break;

        default:
panic("open");
break;
    }

    continue;
} else {

    char fbuf[BUFLEN];
    char res[BUFLEN];
    char *tmp;
    char c_type_header[BUFLEN];
    int nbytes;
    char c_length_header[BUFLEN];

WRITE_OR_CONT(csd, HTTP_STATUS_200, strlen(HTTP_STATUS_200));

    /* create header with file size*/
    exec_cmd(filename, DU_ABS, DU,DU_OPTIONS,res);
    tmp = strtok(res,"\t");

    if(tmp != NULL){
strcpy(c_length_header,"Content-Length: ");
strcat(c_length_header, tmp);
WRITE_OR_CONT(csd, c_length_header, strlen(c_length_header));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));
    }

    /* create header with content type*/
    exec_cmd( filename, FIND_ABS, FIND, FIND_OPTIONS,res);
    tmp = strpbrk(res, " ");

    if (tmp == NULL){
WRITE_OR_CONT(csd, HTTP_CONTENT_TYPE, strlen(HTTP_CONTENT_TYPE));
    } else{
strcpy(c_type_header, "Content-Type:");
strcat(c_type_header, tmp);

```

```

WRITE_OR_CONT(csd, c_type_header, strlen(c_type_header));
    }

    if ((strcasecmp(method, HTTP_METHOD_GET) == 0)){
WRITE_OR_CONT(csd, CHUNKED_ENCODING, strlen(CHUNKED_ENCODING));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));

/* chunked transfer*/
do {
    if ((nbytes = read(file, fbuf, BUFLen)) < 0)
        panic("read");

    sprintf(chunk_size,"%x", nbytes);
WRITE_OR_CONT(csd,chunk_size, strlen(chunk_size));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));
WRITE_OR_CONT(csd, fbuf, nbytes);
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));

} while (nbytes == BUFLen);

WRITE_OR_CONT(csd, LAST_CHUNK, strlen(LAST_CHUNK));
WRITE_OR_CONT(csd, HTTP_NEWLINE, strlen(HTTP_NEWLINE));

    }
}

close(sd);
return 0;
}

```

3. [2 Punkte] Erweitere den Webserver um die Request-Methode HEAD.
Lösung: Siehe HEAD Codeteil in der kompletten Lösung.
4. [4 Punkte] Das Verzeichnis `geheim` im Dateisystem des Webservers wurde hinterhältigweise in `topsecret` umbenannt! Damit die Dateien im Verzeichnis `geheim` auch weiterhin erreichbar sind, soll der Server eine entsprechende Weiterleitung bereitstellen.
 Beispiel: Fordert ein Client die Datei `/geheim/masterplan.txt` an, so antwortet der Server mit Statuscode 301 und leitet den Client zur neuen URI `/topsecret/masterplan.txt` weiter. Programmiere eine entsprechende Erweiterung für den Web-Server!
Lösung: Siehe `uri redirect` Codeteil in der kompletten Lösung.
5. [5 Punkte] Erweitere den Webserver, so dass Antworten im Modus „Chunked Transfer“ gesendet werden. In diesem Modus wird der Rumpf der Nachricht in kleinen Blöcken einer gewählten Größe an den Client geschickt. Dem Client wird dieser Modus durch den Header

Transfer-Encoding: chunked

angekündigt. Vor jedem Block wird jeweils die Länge des Blockes in Bytes (hexadezimal) angegeben. Ein Block der Länge Null signalisiert das Ende des Rumpfes.

Lösung: Siehe `chunked transfer` Codeteil in der kompletten Lösung.

6. [optional, 8 Bonuspunkte] Der Unix-Befehl „`file -i`“ findet heraus, welchen MIME-Typ eine Datei hat¹. Benutze `file`, um den Header `Content-Type` in der Antwort des Webserver richtig zu setzen. Setze außerdem den Header `Content-Length` entsprechend der Länge der übertragenen Datei!

Lösung: Siehe `exec.cmd` Codeteil in der kompletten Lösung.

Hinweis: Löse die Aufgaben in einem Team von zwei bis drei Teilnehmern! Sende deine Lösung als E-Mail mit dem Betreff „Abgabe-Blatt2“ an `singh@informatik.uni-tuebingen.de`. Bitte füge deiner E-Mail eine Liste aller Teammitglieder (jeweils Vor- und Nachname, Matrikelnummer und E-Mail-Adresse) bei. Deine Abgabe soll *nur den Quellcode* beinhalten, den du *unkomprimiert* als Anhang der Mail beifügen solltest!

¹Dies gilt zumindest für Linux- und FreeBSD-Systeme.