Im Rahmen des Projekts entwickeln Sie einen kleinen Webserver für Linux. Seine endgültige Fassung wird die folgenden Features besitzen:

- Frei wählbares Document-Root-Verzeichnis (Standard: `/tmp`)
  und frei wählbarer Port (Standard: 8080)
- `gzip`-Kompression, wenn der Client diese unterstützt (vgl. `mod_deflate` bei Apache), siehe http://httpd.apache.org/docs/2.2/mod/mod_deflate.html
- beliebig viele parallele Verbindungen (Maximalzahl konfigurierbar), über Threads realisiert
- Server beendet sich, wenn das Dokument `/EXIT` angefordert wird
- Support für die Option *Connection: Keep-Alive*, siehe http://en.wikipedia.org/wiki/HTTP_persistent_connection ; Disconnect nach zwei Minuten Inaktivität
- Anzeige des Server-Status, wenn das Dokument `/STATUS` angefordert wird
- Pfadüberprüfung gegen ../../-Attacken
- Aktive Inhalte (auf Server-Seite ausgeführte Skripte)

Die Entwicklung erfolgt in mehreren Stufen. Zu jeder gelösten Teilaufgabe speichern Sie den aktuellen Status des Programms als `projektXX.c` und legen ein Log zum Probelauf als `projektXX.log` bei. (Das kann z. B. Debug-Meldungen direkt aus dem Programm oder eigene Kommentare zum Laufverhalten enthalten.) Bitte erzeugen Sie keine Word-, OpenOffice-, PDF- oder sonstige Dateiformate, die einen speziellen Viewer benötigen; die Log-Dateien sollen einfache Textdateien sein.

Hinweis: Sie dürfen für das Programmieren Zweiergruppen bilden, gegen Ende der Veranstaltung muss aber jede/r einen eigenen Vortrag halten. Sie können die Aufgaben auch unter OS X bearbeiten.

P1. Echo-Programm

Entwickeln Sie ein kleines Echo-Programm, das eine TCP-Verbindung auf Port 8080 annimmt, einen Puffer (teilweise) mit empfangenen Daten füllt und diese Daten wieder an die Gegenstelle zurückschickt; danach soll die Verbindung abgebrochen werden.

Sie brauchen dazu

a) ein paar symbolische Konstanten und Variablen, u. a.:

```
#define SOCKADDR_SIZE sizeof(struct sockaddr_in)
#define BUFLEN        1024
#define PORT          8080
int                sd, conn;            // Socket Descriptors
struct sockaddr_in server, client;      // Socket Addresses
char               buf[BUFLEN];         // Buffer for reading/writing
```

b) die Belegung der Adress-Struktur mit

```
server.sin_port        = htons(PORT);    // Host-to-Network-Konvertierung
server.sin_addr.s_addr = INADDR_ANY;     // IP-Adresse: egal
server.sin_family      = AF_INET;        // Address Family, Internet
```

c) und einen mit

```
sd = socket(PF_INET, SOCK_STREAM, 0);
```

geöffneten Socket, den Sie mit

```
bind (sd, (struct sockaddr*)&sock, SOCKADDR_SIZE);
```

an Port 8080 binden.

d) Über `listen (sd, 0);` müssen Sie diesen in einen empfangsbereiten Zustand schalten,

e) dann können Sie mit

```
int clilen = SOCKADDR_SIZE;
conn = accept (sd, (struct sockaddr *) &client, &clilen);
```

auf eine Verbindungsanfrage warten.

f) Steht die Verbindung, können Sie `conn` wie einen File Descriptor verwenden. Nutzen Sie dann (wie beim Dateizugriff) die Funktionen `read()` und `write()`, um zunächst in den Buffer zu „lesen“ (Daten empfangen) und dessen Inhalt dann wieder zu „schreiben“ (Daten senden).

g) Zum Abbauen der Verbindung rufen Sie vor `close(conn);` noch `shutdown(conn, SHUT_RDWR);` auf. Gleiches gilt für `sd`. (Es gibt *zwei* Socket-Deskriptoren!)

Die Aufrufe von `socket()`, `bind()`, `listen()` und `accept()` können fehlschlagen. Bauen Sie passenden Code ein, der Fehlerfälle abfängt und darauf sinnvoll reagiert – je nach Situation kann eine sinnvolle Reaktion das Beenden des Programms oder ein neuer Versuch sein.

Die Funktionen `htons()` (**h**ost **to** **n**etwork) und `ntohs()` (**n**etwork **to** **h**ost) konvertieren Port-Nummern zwischen Host- und Netzwerk-Darstellung; im Netz wird eine andere Kodierung als auf dem Linux-PC verwendet. (Es geht dabei um die Byte-Order, also: ob zunächst die hochwertigen oder niedrigwertigen Bytes gespeichert werden.) Es gilt z. B.: `htons(256)=1`, `htons(1)=256`; hexadezimal: `01 00` ↔ `00 01`.

Speichern Sie Ihr Programm als `projekt01.c` und dokumentieren Sie das Laufverhalten in einer Protokolldatei `projekt01.log`. Für Tests bauen Sie bei laufendem Programm in einem weiteren Terminalfenster mit `telnet localhost 8080` eine Verbindung zu Port 8080 auf, die von Ihrem Programm entgegen genommen wird. Schreiben Sie dann im Terminal eine Zeile Text (mit [Eingabe] bestätigen), diese sollte anschließend wieder ausgegeben werden.

Hinweis: Wenn Sie kurz nacheinander mehrere Tests durchführen, funktioniert das Programm im zweiten oder späteren Durchlauf evtl. nicht. Das liegt daran, dass TCP-Ports noch eine Weile nach dem Programmende reserviert bleiben und Ihr Programm dann beim erneuten Ausführen Port 8080 nicht binden kann. Hier hilft nur Abwarten, alternativ können Sie vorübergehend auf einen anderen Port ausweichen.

P2. HTTP-Header

Im nächsten Schritt machen Sie Ihr Programm HTTP-kompatibel. An der grundsätzlichen Funktion ändern Sie nicht viel, packen aber einige HTML-Tags um die übertragenen Daten herum und ergänzen einen HTTP-Header.

a) Modularisieren Sie zunächst den bisherigen Code, um das Programm übersichtlich zu halten:

Schreiben Sie Funktionen

```
int setup_port (int port);        // returns: socket descriptor;
                                  // calls socket(), bind(), listen()
int handle_connect (int connfd);  // handles one connection
```

welche die bisher in `main()` erledigten Aufgaben übernehmen. Das Hauptprogramm soll danach im Wesentlichen nur noch aus den Zeilen

```
sd = setup_port (PORT);
conn = accept (sd, (struct sockaddr *) &client, &clilen);
if (conn != -1)
  handle_connection (conn);
shutdown (sd, SHUT_RDWR);
close (sd);
```

bestehen.

b) Verwenden Sie zunächst `dup2()`, um Ausgaben auf der Standardausgabe (File Descriptor `1`) auf den Socket Descriptor `conn` umzuleiten; danach werden alle z. B. mit `printf()` geschriebenen Daten über die Netzwerkverbindung verschickt.

Wenn Sie im Programm noch Meldungen auf der Konsole ausgeben wollen (z. B. für Debugging-Zwecke), können Sie dafür die Standardfehlerausgabe (File Descriptor 2) verwenden (via `write (2, s, strlen(s));` oder `fprintf (stderr, s);`).

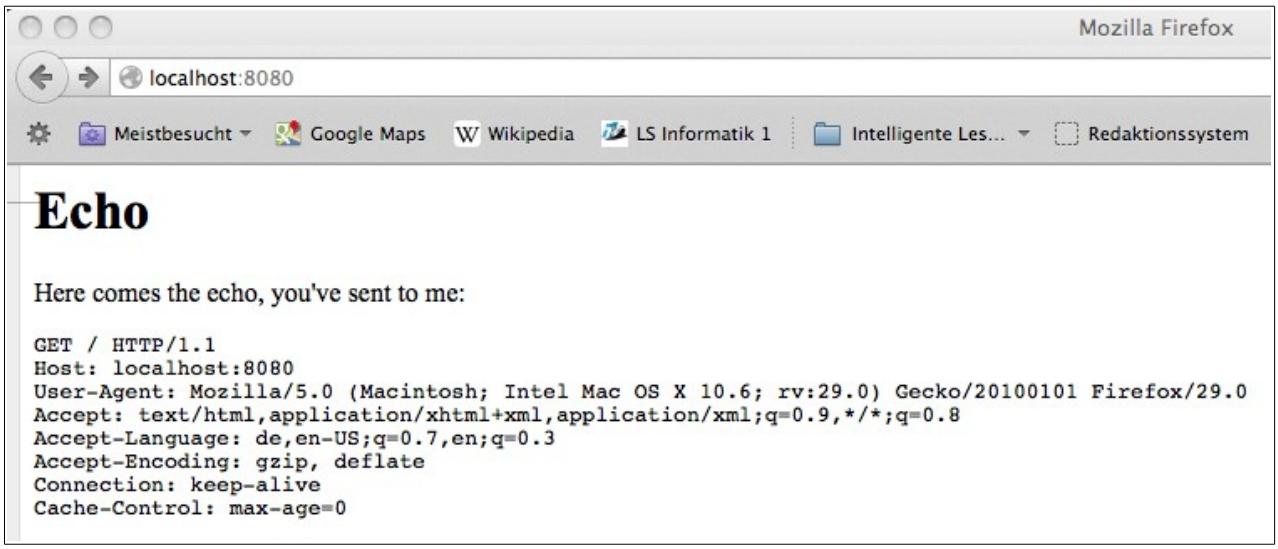
c) Anstatt den Buffer-Inhalt mit `write(conn, ...)` zu schreiben, verwenden Sie nun `printf()` und nutzen als Format-String

```
"HTTP/1.0 200 OK\nContent-Type: text/html\n\n<html><body><h1>Echo</h1>"
"<p>Hier kommt das Echo:</p><pre>%s</pre></body></html>\n"
```

Wichtig: Wenn Sie `printf()` in dieser Form verwenden, müssen Sie danach `fflush(stdout);` ausführen, ansonsten puffert das System eventuell die Daten.

d) Achten Sie darauf, beim Programmende nicht nur `close(conn)`, sondern auch `close(1)` aufzurufen. (Der `shutdown()`-Aufruf muss *nicht* zweimal erfolgen.)

Zum Testen können Sie in einem Webbrowser die Adresse http://localhost:8080 eingeben, es sollte dann eine Seite wie in der folgenden Abbildung erscheinen. Sie erkennen darin den Inhalt der HTTP-Anfrage, welche der Browser (im Beispiel: Firefox) generiert und an den Server sendet.



Speichern Sie Ihr Programm als `projekt02.c` und dokumentieren Sie das Laufverhalten in einer Protokolldatei `projekt02.log`.

P3. Dateien übertragen

In dieser Aufgabe setzen Sie die Kernfunktion eines Webservers um: den Zugriff auf Dateien. Die vom Browser an den Server übertragenen Daten enthalten mehrere, durch Zeilenumbrüche getrennte Header (z. B.: `Host: localhost:8080`), in der ersten Zeile steht aber ein `GET`-Kommando der folgenden Form:

```
GET / HTTP/1.1
```

Der Schrägstrich zwischen `GET` und `HTTP/1.1` ist dabei das angeforderte Dokument (in diesem Fall einfach `/`, was ein Webserver je nach Konfiguration z. B. in `/index.html` oder `/index.php` übersetzt). Verwenden Sie probeweise in einem neuen Testdurchlauf die URL `http://localhost:8080/testdatei.html`, Sie sollten dann sehen, wie sich die GET-Anfrage in

```
GET /testdatei.html HTTP/1.1
```

ändert. Die HTTP-Protokollversion (im Beispiel 1.1) können Sie ignorieren.

a) Parsen Sie aus der `GET`-Zeile im Puffer den vollen Pfadnamen (in den Beispielen `/` bzw. `/testdatei.html`). Sie können davon ausgehen, dass gültige Requests immer direkt mit `"GET "` beginnen, danach folgt der Pfad, der durch ein Leerzeichen terminiert ist (also selbst kein Leerzeichen enthalten darf).

b) Legen Sie an beliebiger Stelle einen Ordner `htdocs` an (z. B. im Code-Verzeichnis, in dem Sie das Programm entwickeln) und bestimmen Sie den absoluten Pfad zu diesem Ordner, etwa `/home/xyz/syspro/projekt/htdocs`. Kopieren Sie die Dateien `index.html`, `page2.html`, `bild1.png` und `bild2.png` aus dem Aufgabenarchiv `sp-ss2014-projekt1.tgz` (siehe Webseite) in diesen Ordner.

c) Definieren Sie im Programm die symbolische Konstante `DOCROOT`; sie soll den vollen Pfad (siehe Aufgabe **b)**, ohne abschließenden `/`) enthalten. Setzen Sie (nach dem Parsen des angeforderten Pfadnamens) aus `DOCROOT` und dem Pfadnamen einen neuen String zusammen. (Dafür können Sie `strncpy()` und `strncat()` verwenden.) Dieser sollte nun bei einer passenden Anfrage (etwa `/index.html`) der korrekte Pfad zur Index-Datei sein.

d) Wie schon in Aufgabe P2 müssen Sie zunächst einen HTTP-Header an den Browser schicken, hier eignet sich der String `"HTTP/1.0 200 OK\nContent-Type: text/html\n\n"`.

e) Öffnen Sie die Datei und „kopieren“ Sie diese (durch Schreiben auf die Standardausgabe) in den verbundenen Netzwerk-Socket. Falls sich die Datei nicht öffnen lässt, geben Sie stattdessen einen Fehlercode aus, im Header muss dann in der ersten Zeile die Fehlermeldung `"HTTP/1.1 404 Not Found\n"` statt `"HTTP/1.0 200 OK\n"` stehen.

Für das Kopieren werden Sie eine Schleife benötigen. Den schon für die Anfrage genutzten Buffer können Sie recyceln und immer wieder `BUFLEN` Bytes aus der Datei lesen (bis `read()` Ihnen `0` zurück gibt).

Auch hier ist wieder zu beachten: Wenn Sie `printf()` und `write(1, ...)` im Wechsel benutzen, müssen Sie nach jedem `printf()`-Aufruf die Standardausgabe flushen (`fflush(stdout);`).

f) Testen Sie das Programm, indem Sie im Browser gültige und ungültige URLs (z. B. `http://localhost:8080/index.html` und `http://localhost:8080/ungueltig.html`) eingeben. (Nach jedem Test müssen Sie das Programm neu starten.)

g) Wenn das funktioniert, erweitern Sie Ihr Programm noch um eine Schleife, die nach jeder bearbeiteten Anfrage auf neue Anfragen wartet – dann kann der Server (nacheinander) beliebig viele Anfragen ohne Neustart beantworten. Überlegen Sie sich vorher, ob der Aufruf von `setup_port()` in die Schleife oder vor die Schleife gehört.

Speichern Sie Ihr Programm als `projekt03.c` und dokumentieren Sie das Laufverhalten in einer Protokolldatei `projekt03.log`.

Heben Sie Ihre Lösungs- und Protokolldateien auf – Sie brauchen Sie am Kursende für die Einreichung.

Überblick

Bei den folgenden Terminen erhalten Sie weitere Aufgaben, welche die Entwicklung des Webservers fortsetzen, darunter (voraussichtlich):

- P4. Simultane Verbindungen und Threads
- P5. Server beenden (`/EXIT`)
- P6. Statistik ausgeben (`/STATUS`)
- P7. gzip-Kompression
- P8. Konfigurierbarkeit
- P9. Pfadüberprüfung
- P10. Keep-Alive
- P11. Aktive Inhalte