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

- Frei wählbare Ports für Proxy-Dienst (Standard: 8080) und Steuerung/Statusabfrage (8081)
- beliebig viele parallele Verbindungen (Maximalzahl konfigurierbar), über Threads realisiert
- Server beendet sich, wenn das Dokument `/exit` angefordert wird (via Status-Port)
- 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 (via Status-Port)
- Caching der geladenen Seiten im RAM, prüfen der Aktualität mit HEAD
- nach Wahl eine von mehreren Zusatzfunktionen (Details folgen später)

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 Log kann z. B. Debug-Meldungen direkt aus dem Programm oder eigene Kommentare zum Laufverhalten enthalten.) Bitte erzeugen Sie keine Word-, LibreOffice-, PDF- oder sonstige Dateiformate, die einen speziellen Viewer benötigen; die Log-Dateien sollen einfache Textdateien sein.

Teilen Sie die Code-Dateien nicht in mehrere einzelne Dateien auf. Wenn Sie bei der Entwicklung eine solche Aufteilung bevorzugen, machen Sie diese für die Abgabeversion rückgängig.

Hinweis: Sie dürfen für das Programmieren Zweiergruppen bilden, gegen Ende der Veranstaltung muss aber jede/r einen eigenen Vortrag halten. Sie können die Aufgaben wahlweise unter Linux oder 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 PORT1         8080
#define PORT2         8081
int                sd1, sd2, conn;            // Socket Descriptors
struct sockaddr_in server1, server2, client;  // Socket Addresses
char               buf[BUFLEN];               // Buffer for reading/writing
```

b) die Belegung der Adress-Struktur mit

```
server1.sin_port        = htons(PORT1);       // Host-to-Network-Konvertierung
server1.sin_addr.s_addr = INADDR_ANY;         // IP-Adresse: egal
server1.sin_family      = AF_INET;            // Address Family, Internet
```

c) und einen mit

```
sd1 = socket(PF_INET, SOCK_STREAM, 0);
```

geöffneten Socket, den Sie mit

```
bind (sd1, (struct sockaddr*)&server1, SOCKADDR_SIZE);
```

an Port 8080 binden.

d) Über `listen (sd1, 0);` müssen Sie diesen in einen empfangsbereiten Zustand schalten,

e) dann können Sie mit

```
int clilen = SOCKADDR_SIZE;
conn = accept (sd1, (struct sockaddr *) &client, &clilen);
```

auf eine Verbindungsanfrage warten.

f) Steht die Verbindung, können Sie den Socket Descriptor `conn` wie einen File Descriptor verwenden, also (wie beim Dateizugriff) die Funktionen `read()` und `write()` nutzen, 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 `sd1`. (Es gibt *zwei* Socket-Deskriptoren!)

h) Ersetzen Sie nach ersten Tests die Aufrufe von `read()` und `write()` durch entsprechende Aufrufe von `recv()` und `send()`, wie in Foliensatz 9, Folie 15 beschrieben.

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.

Nach Programmfehlern kann es vorkommen, dass der verwendete Port noch ein paar Sekunden gesperrt ist, so dass ein direkter Programmneustart nicht möglich ist. (Der `bind()`-Aufruf schlägt dann fehl.)

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, implementieren damit also (provisorisch) eine Art Webserver. 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

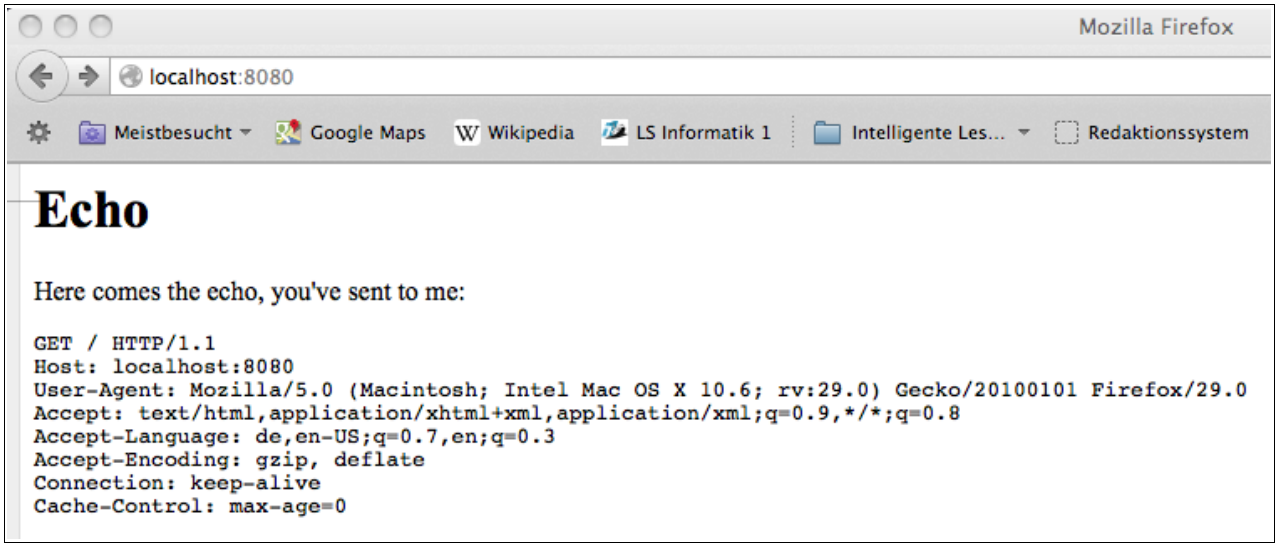
```
sd = setup_port (PORT);
conn = accept (sd, (struct sockaddr *) &client, &clilen);
if (conn != -1)
  handle_connection (conn);
shutdown (sd, SHUT_RDWR);
close (sd);
```

bestehen.

b) Um formatierte Informationen über den Socket zu verschicken, bauen Sie in einem Buffer `buf` eine Nachricht mit `sprintf (buf, format, ...);` zusammen und verschicken diese dann mit `send (conn, buf, strlen(buf), 0);`. Als Format-String verwenden Sie

```
"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"
```

Zum Testen können Sie in einem Browser die Adresse `http://localhost:8080` eingeben, es sollte 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. Zweiten Port aktivieren

Jetzt geht es darum, den zweiten Port (standardmäßig: 8081) zu aktivieren. Dazu verwenden Sie den zweiten Socket `sd2` und binden ihn an diesen zweiten Port. Sie können dafür die vorhandene Funktion `setup_port` einfach ein zweites Mal aufrufen, wenn Sie diese so anpassen, dass Sie ihr auch den Server übergeben – die neue Signatur ist also

```
setup_port (int port, struct sockaddr_in *server);
```

Im Hauptprogramm (`main`) soll nun nach

```
sd1 = setup_port (PORT1, &server1);
sd2 = setup_port (PORT2, &server2);
```

eine Anfrage an einen der beiden Ports entgegen genommen werden. Sie können dazu aber nicht einfach zwei `accept()`-Aufrufe hintereinander schreiben (weil schon der erste blockiert, bis eine Anfrage kommt) – stattdessen müssen Sie ein `fd_set` anlegen und mit `select()` arbeiten (vgl. Foliensatz 6, Folien 13–19). Wenn `select()` zurück kehrt, müssen Sie mit zwei Fallunterscheidungen prüfen, ob auf `sd1` oder auf `sd2` eine Anfrage anliegt – abhängig davon, auf welchem Port die Anfrage kommt, führen Sie dann `accept()`, `handle_connection()`, `shutdown()` und `close()` für `sd1` oder `sd2` aus.

Prüfen Sie, dass Ihr Programm korrekt reagiert, wenn Sie einen der beiden Ports (8080, 8081) via Telnet oder aus dem Browser heraus kontaktieren.

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

P4. Simultane Verbindungen

Im nächsten Schritt der Proxy-Implementierung geht es darum, mehrere simultane Verbindungen zu erlauben. Hier kommen die Threads ins Spiel.

a) Passen Sie zunächst das Programm so an, dass es nicht nur eine einzige Verbindung annimmt, sondern in einer Endlosschleife immer wieder `select()`, `accept()` (für Port 8080 oder 8081) und `handle_connection()` aufruft. Vorsicht: Das `fd_set` müssen Sie vor jedem Aufruf von `select()` neu initialisieren (`FD_ZERO`, `FD_SET`), weil im letzten Durchgang `select()` die Inhalte verändert hat.

Die Socket-Deskriptoren `sd1` und `sd2` dürfen jetzt nicht mehr geschlossen werden. (Sie können die `shutdown()`- und `close()`-Aufrufe für `sd1` und `sd2` aber provisorisch hinter die Endlosschleife verschieben, weil eine spätere Version des Programms auch einen ordentlichen Ausstieg erlauben wird.)

b) Für die Parallelisierung erzeugen Sie unmittelbar nach jedem `accept()`-Aufruf einen neuen Thread, der sich um diese Verbindung kümmern soll. Das Hauptprogramm soll dabei den Überblick über die offenen Verbindungen behalten, dazu legen Sie eine Tabelle an (deren Datenstruktur Sie sich selbst überlegen müssen). Insgesamt wird das ungefähr so aussehen:

```
#define MAX_SERVERS ...
struct server {
  ...   // Was müssen Sie für jeden speichern?
};
struct server servers[MAX_SERVERS] = { 0 };
```

Zwischen `accept()` und Thread-Erzeugung können Sie dann einen neuen Tabelleneintrag anlegen. Vermerken Sie in der `server`-Struktur auch, ob die Verbindung über Port 8080 oder 8081 aufgenommen wurde. (Die eigentliche Proxy- und Steuerungslogik, die über diese beiden Ports getrennt verfügbar ist, implementieren Sie später.)

c) Beendete Threads müssen Sie auch wieder einsammeln. Der Haupt-Thread (in dem die Schleife mit `accept()` läuft) ist dafür nicht geeignet, weil er evtl. sehr lange blockiert, bis eine neue Anfrage eintrifft. Stattdessen benötigen Sie einen weiteren Thread, der nur eine Aufgabe hat: Er sucht in der Tabelle nach fertigen Threads (der Tabelleneintrag muss beim Beenden des Threads entsprechend gekennzeichnet werden), sammelt den zugehörigen Thread ein und setzt den Tabelleneintrag auf „frei“ zurück. Hier gibt es ein Synchronisationsproblem: Der Haupt-Thread könnte gleichzeitig mit dem neuen Einsammel-Thread auf die Tabelle zugreifen. Darum schützen Sie die Tabelle mit einem POSIX-Thread-Mutex. Diesen müssen Sie definieren, initialisieren und an geeigneten Stellen „locken“ (`pthread_mutex_lock`) und „unlocken“ (`pthread_mutex_unlock`).

Achten Sie auch darauf, nicht mehr verwendete Tabelleneinträge wieder freizugeben, damit sie bei der nächsten Anfrage verwendet werden können.

d) Ihr Kollektor fragt vermutlich in einer Endlosschleife immer wieder die Einträge `servers[i].finished` (für alle `i`) ab – hier gehe ich davon aus, dass Ihre Datenstruktur ein Feld `finished` besitzt, das Sie auf `false` initialisieren und nach dem Abbruch der Verbindung auf `true` setzen. Eine solche Endlosschleife ist sehr rechenintensiv, und dieser Thread verbraucht damit unsinnig viel CPU-Zeit. Besser wäre es, wenn der Kollektor überwiegend schlafen würde. Sie können dieses Verhalten mit einem weiteren Mutex realisieren: Der Kollektor lockt (am Anfang der Endlosschleife) den neuen Mutex, die Freigabe erfolgt im Server-Thread (unmittelbar, bevor er sich beendet). Bauen Sie auch den Einsatz dieses zweiten Mutex (der z. B. `kollektor_mutex` heißen könnte) in das Programm ein.

Für diese Aufgaben können Sie (z. B.) Funktionen mit den folgenden Prototypen verwenden:

```
void *handle_connection (void *arg);
void *collector (void *none);
int create_server_thread (int connfd);
```

Beachten Sie bei der Parameterübergabe an Threads (via `pthread_create`), dass hier ein Pointer für die Parameter erwartet wird; Sie könnten z. B. die Adresse des Tabelleneintrags in der Form

```
(void*) &servers[index]
```

an die Thread-Funktion `handle_connection()` übergeben.

Speichern Sie Ihr Programm als `projekt04.c` und dokumentieren Sie das Laufverhalten in einer Protokolldatei `projekt04.log`.