P4. Simultane Verbindungen

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

a) Unmittelbar nach jedem `accept()`-Aufruf erzeugen Sie einen neuen Thread, der sich um diese Verbindung kümmern soll. Das Hauptprogramm soll dabei aber 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.

b) 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.

c) 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`.

(Zur groben zeitlichen Orientierung: Sie sollten beim heutigen Termin mit der Bearbeitung von Aufgaben P1–P3 fertig werden und von der neuen Aufgabe P4 mindestens Teil a) bearbeiten.)