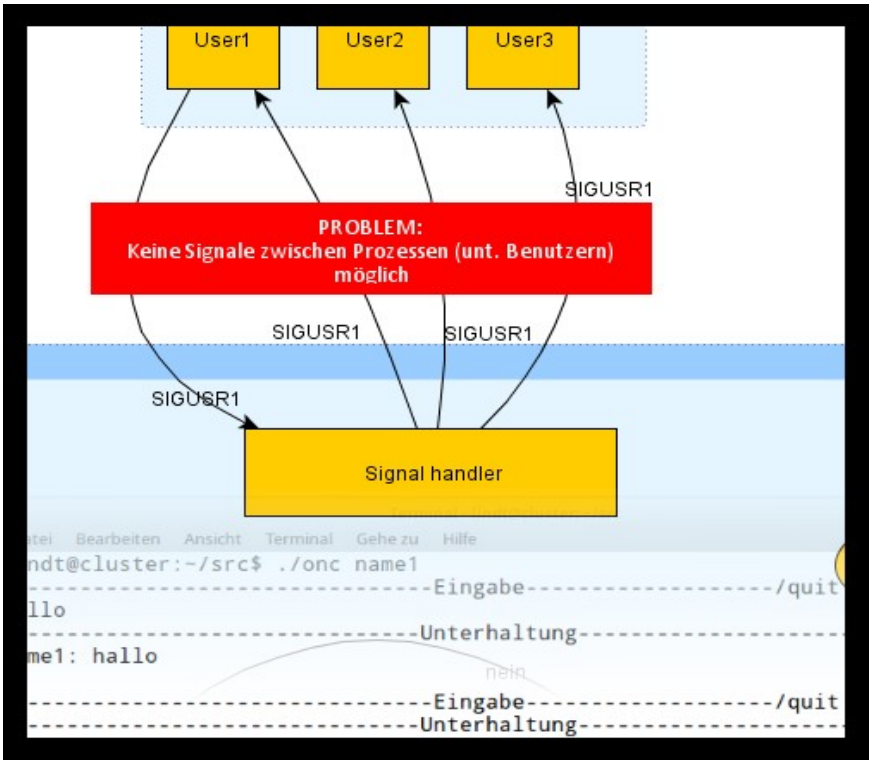
Lokales Chatsystem



Bericht zum Praktikum:

Betriebssystemnahe C-Programmierung

von: Paul Lindt, Irina Lindt

Inhaltsverzeichnis

Aufgabenstellung:

Das Praktikum sollte dem Kennenlernen der betriebssystemnahen Programmierung unter LINUX/UNIX dienen. Dabei sollten die erworbenen Kenntnisse zur Entwicklung eines einfachen Programms eingesetzt werden.
Unser Ziel bei diesem Übungsprogramm war es, ein lokales Chatsystem zu entwickeln, welches die Kommunikation mehrerer User auf einem Linuxsystem, insbesondere einem Cluster, ermöglicht. Ursprünglich sollte es mit Shared Memory funktionieren, die Aufgabenstellung wurde allerdings um eine alternative Implementierung mit Sockets erweitert.

Allgemeine Konzepte:

Shared Memory ist ein häufig genutztes Konzept der Interprozess-Kommunikation, welches unterschiedlichen Prozessen den gemeinsamen Zugriff auf ein Stück Speicher ermöglicht.

Sockets sind ein aus BSD stammendes Konzept zur Netzwerkkommunikation. Dabei wird dem Entwickler die Möglichkeit gegeben, über einen File Deskriptor mit dem externen Programm zu kommunieren.

Threads erlauben die Aufspaltung eines Programms in mehrere Aufführungsstränge, die sich einen gemeinsamen Adressraum teilen, ansonsten aber unabhängig von einander funktionieren.

Vorüberlegungen:

Da der Einsatz von Shared Memory bereits in der Aufgabenstellung festgelegt wurde, konzentrierten sich unsere ersten Überlegungen hierauf. Unter anderem war zunächst die Entscheidung nach der Art der Shared Memory zu treffen. Dabei standen sich die von Solaris abstammende Version und die POSIX-Version gegenüber. Beide boten vergleichbare Funktionalität und sollten inzwischen auf allen Linux-Systemen verfügbar sein. Die Entscheidung fiel schließlich für POSIX - aus folgenden Gründen:

- POSIX ist der weiter verbreitete Standard
- die Identifikation des Speichers durch einen Dateinamen in Verbindung mit dem Einbinden unter /dev/shm sollte einen geringeren Debuggingaufwand ermöglichen
- der Zugriff durch reguläre Schreib- und Lesefunktionen versprach verständlicheren Code

Außerdem musste das Programm über eine Abstraktionsschicht verfügen, die sowohl den Einsatz der Shared Memory als auch von Sockets erlaubte. Schnell entstand die Idee, dass wir eine an den jeweiligen Einsatz angepasste Struktur brauchen, welche die zugehörigen

Verbindungsinformationen enthält. Diese wurde als struct “connection”, im späteren Verlauf struct on_connection realisiert.

```
struct on_connection
{
      int on_socket;
      struct thread_list *server_threads;
      struct socket_list *sockets;

      int client_number;
};
```

Netzwerk

```
struct on_connection
{
      int shm_log_descriptor;
      int shm_sync_vars_descriptor;
      struct on_sync_vars *
                  sync_vars;
      };
```

Shared Memory

Wie man sieht, haben die beiden Strukturen, trotz gleichen Namens und gleicher Funktion in beiden Programmvarianten, auf den ersten Blick vollkommen unterschiedlichen Aufbau. Bei genauer Betrachtung merkt man, dass shm_log_deskriptor und on_socket durchaus beides File-Deskriptoren sind, die den Kommunikationskanal darstellen. Aufgrund der gravierenden Unterschiede bei der Benutzung haben wir uns aber gegen die Benutzung eines einheitlichen Namens für die beiden entschieden. Der Rest sind in beiden Fällen Hilfvariablen für die Organisation der Arbeit.

Ebenfalls sehr früh wurde klar, dass wir eine Aufteilung in einen Server und viele Clients brauchen werden, da wir auch in der Shared Memory einen Prozess brauchen, der als erster gestartet wird, die Shared Memory als Kommunikationskanal erstellt und die Rechte vergibt, so dass andere Prozesse darauf zugreifen können. Wir entschieden uns, den Server ond und den Client onc zu nennen.

Des Weiteren sollten die Clients alle eingegebenen Textnachrichten weiterleiten, andererseits musste es auch eine Möglichkeit geben, die Programme zu verlassen. Deshalb wurde das Kommando /quit eingeführt, welches bei Eingabe das jeweilige Programm beendet.

Ob das Projekt in der Netzwerk- oder der Shared-Memoryversion übersetzt wird, lässt sich durch Prekompileranweisungen steuern. Um genau zu sein, muss das Symbol ON_SHARED_MEMORY definiert sein, damit die on_shared_memory.h datei eingebunden wird. Ansonsten wird die on_network.h benutzt.

Versuch 1: Shared Memory + Signale

Die erste Version des Programms entstand aus der Überlegung, dass wir Shared Memory für den Austausch von Textnachrichten brauchen, allerdings auch eine Möglichkeit, andere Clients über das Absenden von Nachrichten zu informieren, da Shared Memory, anders als Sockets, kein blockierendes read bietet. Signale schienen da die logische Wahl, da sie eine einfache Möglichkeit bieten, eine Reaktion in einem anderen Prozess auszulösen.

Der durch ein Signal ausgelöste Aufruf einer Callbackfunktion sollte dazu genutzt werden, den Inhalt der Shared Memory auf dem Bildschirm zu bringen und dadurch allen Usern bei einer Änderung sichtbar zu machen.
Leider haben Signale einen schwerwiegenden Nachteil. In der Regel ist es nicht möglich, ein Signal an den Prozess eines anderen Users zu schicken. Da gerade die Kommunikation mehrerer Nutzer ermöglicht werden sollte, wurden die Signale vollkommen unbrauchbar.

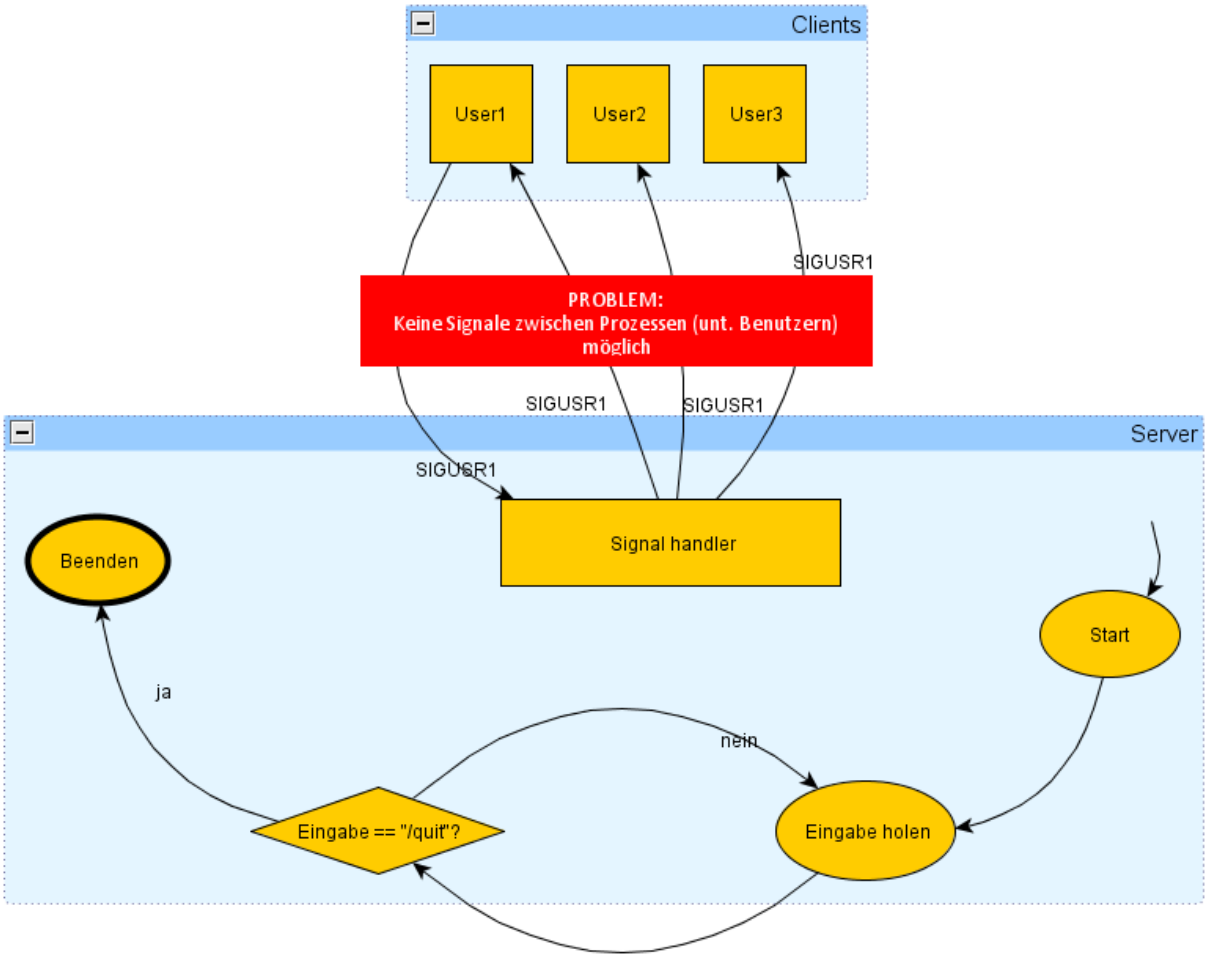


Abbildung 1: Versuch 1 aus Serversicht

Versuch 2: Shared Memory + Signale + Message Queues

Aus der Hoffnung, möglichst viel aus dem ersten Versuch zu retten, entstand die Idee für den zweiten Versuch. Message Queues, die als solche schon gute Kandidaten als IPC System für ein lokales Chatprogramm sind, ermöglichen auch die Kommunikation zwischen Prozessen mehrerer Benutzer. Desweiteren geben sie Prozessen die Möglichkeit, sich durch ein Signal informieren zu lassen, sobald eine Nachricht über eine vorher leere Message Queue verschickt wird.

```
struct sigevent on_sigevent;
      on_sigevent.sigev_signo = SIGUSR1;
      if(mq_notify(mq_descriptor, &on_sigevent) == -1)
            perror("konnte nicht fuer benachrichtigung durch mq
                        registrieren");
```

Die Idee war, die im ersten Versuch vorgesehenen Reaktionen auf Signale wiederzuverwenden und die Signale von den Message Queues statt direkt von den Prozessen auslösen zu lassen. Obwohl das Prinzip grundsätzlich funktionierte, wurde das Programm unnötig kompliziert, unübersichtlich und schließlich verworfen.

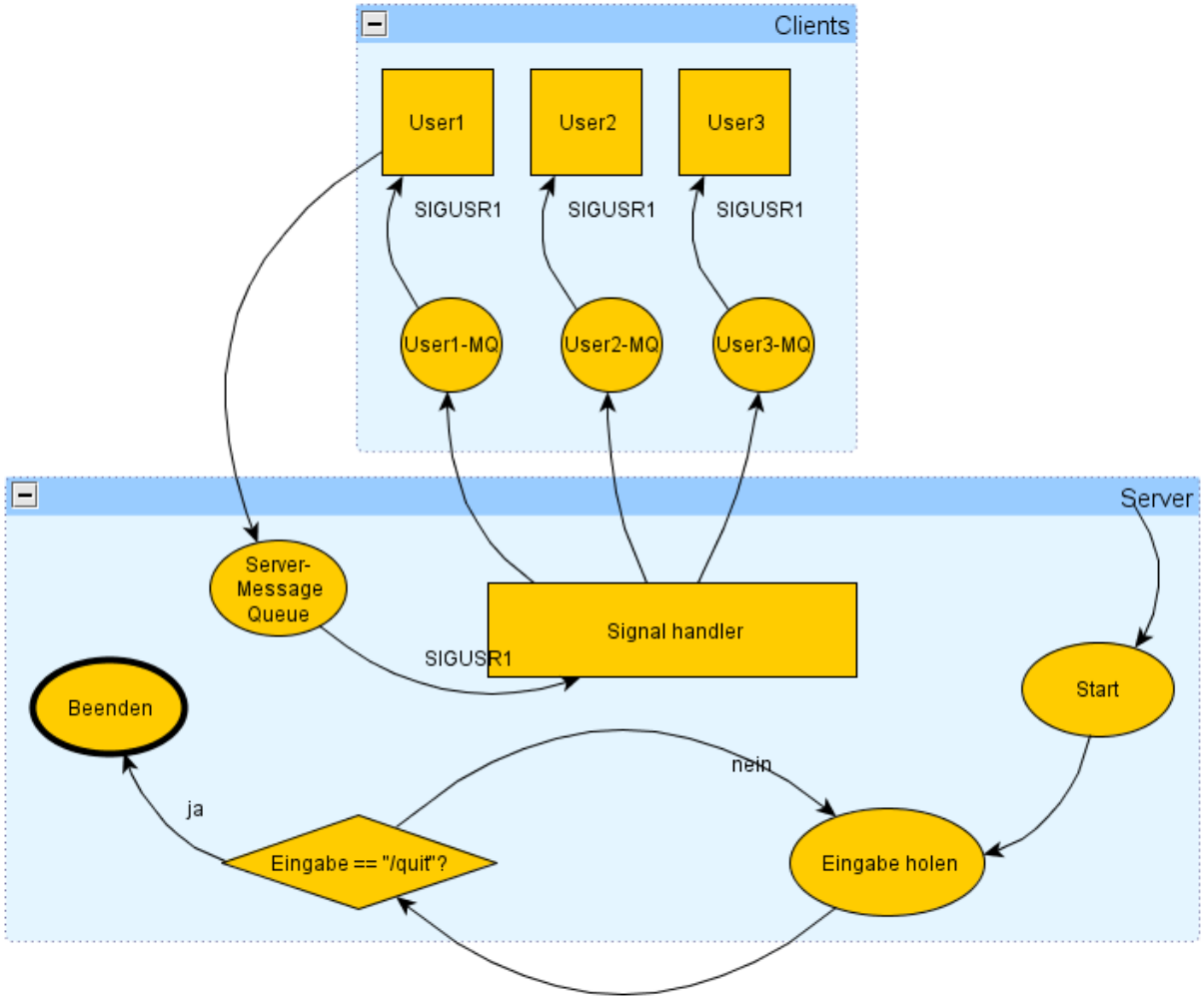


Abbildung 2: Versuch 2 aus Serversicht

Versuch 3: Shared Memory + Condition-Variablen

Beim dritten Versuch wurde das Programm praktisch komplett neu geschrieben, da es ein grundsätzlich anderes Funktionsprinzip benötigte.

“Eine Condition-Variable wird verwendet, indem ein Thread darauf wartet und ein anderer sie Auslöst.” So wird das Prinzip der Condition-Variablen im Tanenbaum beschrieben. Daraus entstand die Idee, die Ein- und Ausgabe von Nachrichten weiterhin getrennt zu behandeln, die Trennung aber nicht mehr durch Signale und Callbackfunktionen, sondern durch Threads zu realisieren. Dieses passiert durch die Abspaltung des server_interface_thread und Weiterführung des Ursprungsthreads als user_interface.

Beide Threads befinden sich dabei in einer Endlossschleife und können in ihrer Ausführung blockiert werden. Während es beim user_interface das Warten auf Benutzereingaben ist, wartet der server_interface_thread auf eine Condition-Variable. Und beide warten darauf, ein Kommando ("/quit") zu bekommen, welches es ihnen erlaubt, die Endlosschleife und auch das Programm zu beenden.

Der Server hat einen vergleichbaren Aufbau. Allerdings ist sein user_interface-Thread nicht in der Lage, Textnachrichten abzuschicken, sondern nur dazu da, auf das Kommando zum Aufhören zu warten. Der ond_server_thread übernimmt die Aufgabe, alle Klienten über gesendete Nachrichten zu informieren.

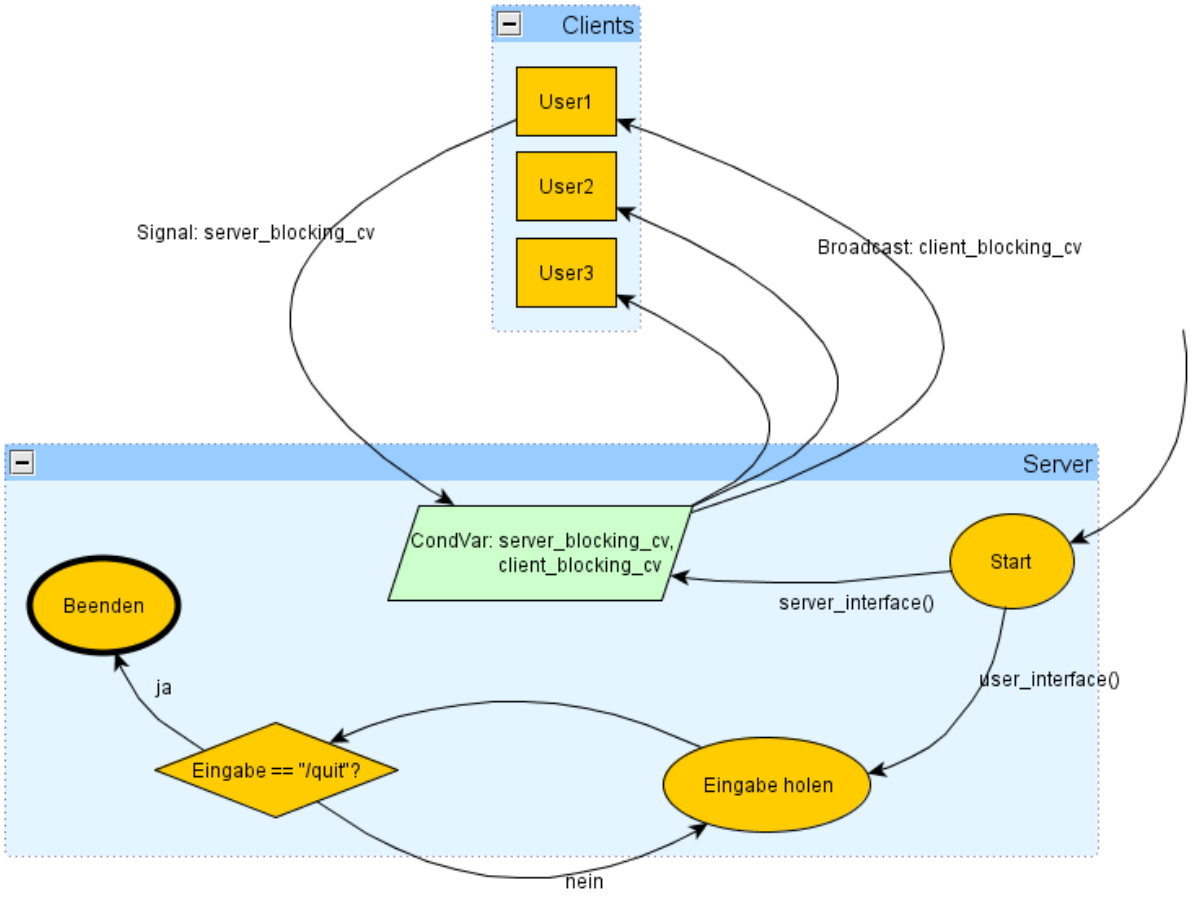


Abbildung 3: Versuch 3 aus Serversicht

Im Programm kommen POSIX-Threads (pthread) zur Anwendung, wodurch die Wahl der POSIX-Shared-Memory im Nachhinein eine zusätzliche Bestätigung bekommen hat.

Das Kernstück des Programms ist die on_sync_vars Struktur, welche die notwendigen Condition-Variablen samt zugehörigen Mutex enthält.

```
struct on_sync_vars
{
      pthread_mutex_t server_mutex, client_mutex, log_mutex;
      pthread_cond_t server_blocking_cv, clients_blocking_cv;
};
```

Dabei blockiert die server_blocking_cv den server_interface_thread, bis sie von einem Client beim Senden einer Nachricht aktiviert wird. Daraufhin wird auch die client_blocking_cv durch ein Broadcast-Signal aktiviert, und gibt dadurch die server_interface_threads der Klienten frei, wodurch diese wiederum allen Benutzern das aktualisierte Chatlog anzeigen.
Dies passiert, indem im server_interface_thread in einer Endlosschleife on_inform_clients aufgerufen wird.

```
void on_inform_clients(struct on_connection * my_connection)
{
      pthread_mutex_lock(&my_connection->sync_vars->server_mutex);
      pthread_cond_wait(&my_connection->sync_vars->server_blocking_cv,
            &my_connection->sync_vars->server_mutex);

      #ifdef DEBUG
      printf("%s\n", "informiere clienten");
      #endif

      pthread_mutex_lock(&my_connection->sync_vars->client_mutex);
      pthread_cond_broadcast(&my_connection->sync_vars->clients_blocking_cv);
      //daraufhin broadcast fuer die andere
      pthread_mutex_unlock(&my_connection->sync_vars->client_mutex);
      pthread_mutex_unlock(&my_connection->sync_vars->server_mutex);

}
```

Da der Austausch der Condition-Variablen durch Mapping der zugehörigen shm-Datei vereinfacht wurde, während für das Nachrichtenlog die write- und read-Operationen besser geeignet waren, werden zwei unterschiedliche Dateien benutzt. Die on_sync_vars dient dem Austausch der Condition-Variablen samt zugehörigem Mutex, sowie ein weiteres Mutex für die Synchronisation des Zugriffs auf die zweite shm-Datei, welche die gespeicherten Nachrichten enthält. Das Mapping im Speicher erlaubt es, die Variablen der Klientprogramme durch einfaches Setzen der entsprechenden Zeiger zu benutzen.

```
if((my_connection->sync_vars = mmap(NULL,       //der Kernel darf die Adresse
bestimmen
                                sizeof(struct on_sync_vars),        //groesse
                                PROT_READ | PROT_WRITE,
                                MAP_SHARED,
                                my_connection->shm_sync_vars_descriptor,
                                0)) == MAP_FAILED)
```

Die Datei /on_log dient dem Austausch der Textnachrichten und wird gleichzeitig als Log benutzt. Da auf ein Mapping dieser Datei im Speicher verzichtet wurde, müssen wir uns auch nicht mehr um ihre Größe kümmern, da sie einfach mitwächst. Da auf diese Datei mehrere Prozesse lesend und schreibend zugreifen, wurde, wie bereits oben erwähnt, das log_mutex, welches in on_sync_vars enthalten ist, für die Absicherung dieser Zugriffe eingeführt.

Für ein beim Testen nicht immer gleich sichtbar werdendes Problem sorgten die Fähigkeit der Mutexe und Condition-Variablen, public oder private zu sein, da, wenn diese nicht explizit auf public gesetzt wurden, immer nur der zuletzt gestartete Client Nachrichten empfangen konnte.

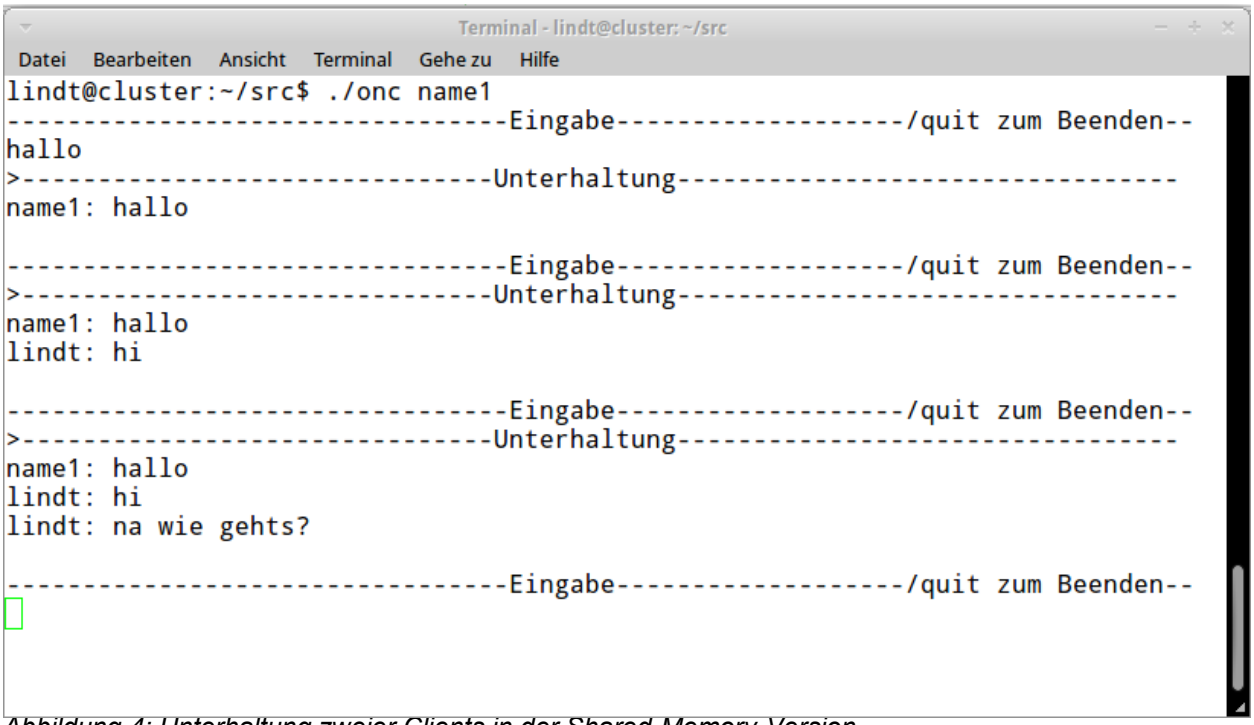


Abbildung 4: Unterhaltung zweier Clients in der Shared-Memory-Version

Alternative Implementierung: Netzwerk

Sockets sind nicht unbedingt die offensichtlichen Mittel der Wahl, wenn man versucht ein lokales Chatsystem zu bauen, und dies stellte uns vor einige Herausforderungen. Während die Kommunikation aus Klientensicht durch die Behandlung von Sockets als Dateien durchaus analog ist, gibt es für den Server einen grundlegenden Unterschied. Er muss nämlich in der Netzwerkversion zum einen zu jedem Klient eine eigene Verbindung halten, daraus folgernd aber auch die Nachrichten zwischen den unterschiedlichen Klienten tatsächlich austauschen. Es reicht ihm auch nicht, die Klienten nur über deren Existenz zu informieren.

Dies führte zur Einführung der socket_list sockets in on_connection der Netzwerkversion, sowie der thread_list, da alle Sockets parallel überwacht werden müssen, so dass für jedes Socket ein eigener Thread notwendig wird.

An dieser Stelle wäre auch eine Implementierung mittels der Funktion select() möglich. Dabei werden die File Deskriptoren der Sockets in einem Set verwaltet, und man wartet mittels der Funktion select() darauf, dass einer der Sockets in diesem Set beschreibbar wird. Dabei müssen keine extra Threads für jedes Socket angelegt werden. Wir entschieden uns jedoch für die zuerst beschriebene Variante.

Da jeder Thread aber sowohl das ihm zugeordnete Socket als auch die gesamte Liste der Sockets kennen musste, um die Nachricht an sie weiterzuleiten, pthread aber nur die Übergabe eines Parameters an den neuen Thread erlaubt, musste eine weitere Struktur her, welche die beiden Informationen, in sich vereinte. So entstand `tmp_socket_struct`

```
struct tmp_socket_struct
{
      int socket;
      struct on_connection * connection;
};
```

Aber auch für die Clients ist die Kommunikation nicht hundertprozentig übertragbar. Einerseits wird die Handhabung enorm vereinfacht, weil, wie bereits erwähnt, auf Sockets blockierendes Lesen möglich ist, wodurch sich sehr einfach “busy waiting” verhindern lässt. Andererseits kann man die Sockets nicht zur Speicherung des Kommunikationsverlaufs einsetzen, wodurch die Netzwerkversion dem Nutzer immer nur die zuletzt empfangene Nachricht ausgibt, während die Shared-Memory-Version alle Nachrichten der Unterhaltung ausgibt.
Das Verhalten ließe sich zwar z.B. durch die Einführung einer Datei, in welcher jeder Client tatsächlich ein Log führt, angleichen, wir fanden aber, dass der heutige Zustand angesichts dessen, dass er einen wichtigen Unterschied zwischen Sockets und Shared Memory verdeutlicht, durchaus akzeptabel ist.

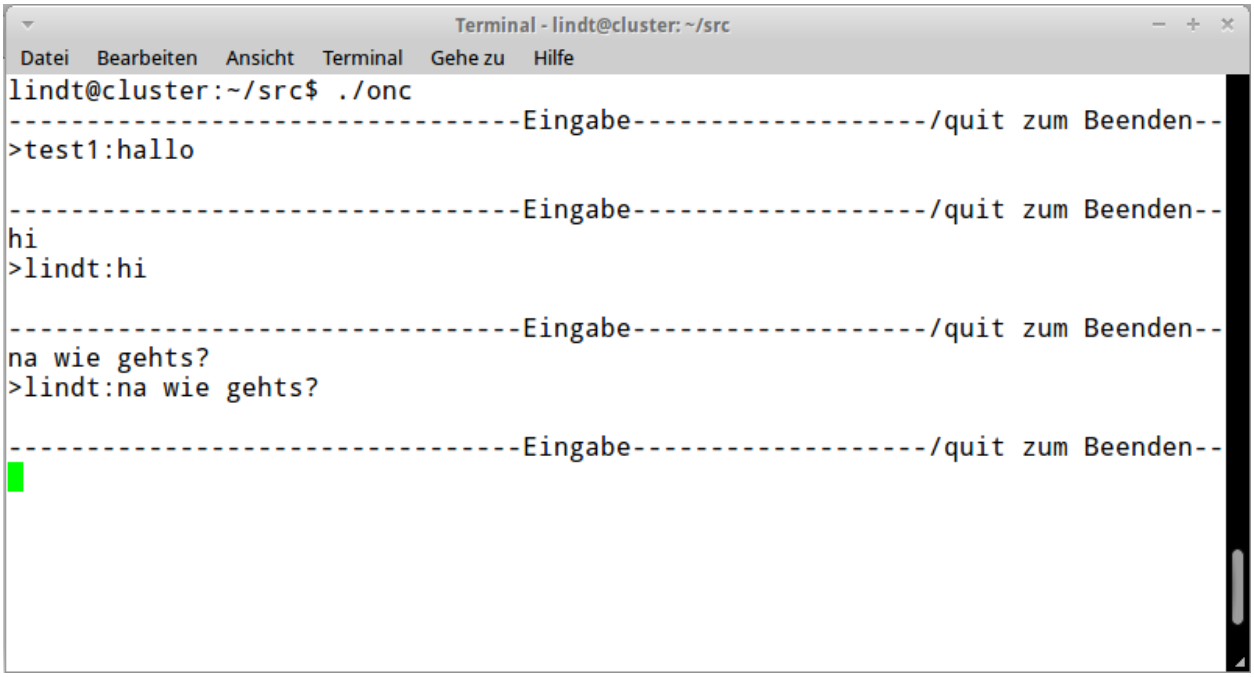


Abbildung 5: Unterhaltung zweier Clients in der Socketversion

Wie die nachfolgende Abbildung zeigt, gibt es aus Sicht des Benutzers auf den ersten Blick keinen Unterschied zwischen der Shared-Memory-Version und der Implementierung mittels Sockets.

Ein gravierender Unterschied zwischen den Implementierungen zeigt sich erst auf den zweiten Blick: das Fehlen von Zugangskontrolle zu der Unterhaltung. Während die Shared-Memory-Version nur für Benutzer des Systems zugänglich ist, ist es in der Netzwerkversion jedem möglich, an der Unterhaltung teilzunehmen.

Was noch fehlt:

Unserem Chatsystem fehlen einige bei weiter entwickelten Lösungen häufig vorhandene Funktionen, insbesondere die Möglichkeit von privaten Chats.

Private Chats wären in der Shared-Memory-Version durchaus möglich durch Wählen eines eigenen Namens für die Shared-Memory-Dateien und anschließendes Setzen der benötigten Rechte, allerdings sahen wir dies als zu aufwendig an.

Fazit:
Wir haben den Umgang mit mehreren IPC-Systemen mit ihren jeweiligen Vorzügen und Grenzen kennengelernt, sowie Erfahrungen mit fortgeschrittenen Konzepten der C-Programmiersprache, wie Callbacks, gesammelt.

Desweiteren wurden Lücken in Soft Skills, insbesondere der Planung, deutlich.

Quellen:

1. *“Moderne Betriebssysteme”* von Andrew S. Tanenbaum in der zweiten Auflage.
2. Manpages: kill(2), signal(2), socket, shm_open, fchmod, pthread_create, getpwent, getuid
3. http://www.cs.cf.ac.uk/Dave/C/node27.html
4. https://computing.llnl.gov/tutorials/pthreads/ , insbesondere die Abschnitte zu Threads und Condition-Variablen
5. http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_condattr_setpshared.html
6. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html