Institut für Informatik

der Ludwig-Maximilians-Universität München

Systempraktikum – Wintersemester 2010/2011

Prof. Dr. Dieter Kranzlmüller
Dr. Nils gentschen Felde, Christof Klausecker, Johannes Watzl

Blatt 7— Vertiefung & Projekt III: Sockets, Signale, Shared Memory, Dateilisten, Suche

Abgabedatum theor. Aufgaben	Abgabedatum prakt. Aufgaben	Deadline Projektaufgaben
—	—	16.01.

Projekt-Aufgaben (Blatt 7)

Aufgabe PROJ-7-1

[Module **`server`**, **`client`**, **`connection`**] Sockets

Um das Projekt nicht nur lokal auf einem Rechner betreiben zu können, müssen die bisher verwendeten Named Pipes in Sockets umgewandelt werden. Schreiben Sie zunächst zwei Funktionen im Modul `connection`, um den Einsatz von Sockets zu erleichtern:

a. `int createPassiveSocket(uint16_t *port);`
Diese Funktion erzeugt einen Socket, zu dem eine Verbindung aufgebaut werden kann. Der Socket ist an einen bestimmten Port `port` gebunden und wartet auf neue Verbindungen (passiv). Verwenden Sie die Funktion `socket()`, um ein verbindungsbasierten Socket (TCP/IPv4) zu erstellen. Binden Sie den Socket dann mittels der Funktion `bind()` an alle Netzadapter des Computers und verwenden Sie den Parameter `port` für den Port. Achten Sie darauf, dass `port` als Zeiger realisiert ist. Im dritten Schritt setzen Sie den Socket mit der Funktion `listen()` horchend. Benutzen Sie schließlich die Funktion `getsockname()`, um den verwendeten Port zu erfahren. und weisen sie diesen Wert dem Zeiger `port` zu. Damit kann man durch Aufruf mit dem Wert 0 für `port` einen automatisch vom System zugewiesenen Port bekommen. Der Rückgabewert der Funktion ist der Filedeskriptor des neu erstellten Sockets oder -1 im Fehlerfall.

b. `int connectSocket(struct in_addr *ip, uint16_t port);`
Diese Funktion wird verwendet, um sich zu einem passiven Socket eines anderen Prozesses zu verbinden. Dieser entfernte Socket wird durch die Adresse `ip` und den Port `port` identifiziert. Verwenden Sie die Funktionen `socket()` und `connect()`, um sich zu einem Socket zu verbinden. Der Rückgabewert ist der neu verbundene Socket oder -1 im Fehlerfall.

c. Stellen Sie jetzt Server und Client auf Sockets statt Named Pipes um. Verwenden Sie die beiden obenstehenden Funktionen und die Werte aus der Konfiguration für IP und Port. Entfernen Sie aus der `struct actionParameters` des `protocol`-Moduls die beiden Pipe-Variablen `s2c` und `c2s`, und führen Sie stattdessen eine neue Variable `int comfd` ein, die den Filedeskriptor des verbundenen Sockets erhält. Über diesen Filedeskriptor kann (im Unterschied zu Pipes) gelesen und geschrieben werden.
Im Server müssen Sie jetzt statt dem blockierenden Erzeugen der Named Pipe den Aufruf `accept()`

[0]Stand: 15. Dezember 2010

vor Ihrer Hauptschleife verwenden. Momentan soll nur eine Verbindung angenommen werden können, Sie können also nach erfolgreichem Aufbau der Verbindung den passiven Socket schließen.

Achten Sie darauf, dass die Struktur `struct sockaddr_in` den Port in "Network Byte Order" erwartet. Sehen Sie sich dazu die Funktionen `htons()` und `ntohs()` an und wandeln Sie den Parameter `port` dementsprechend um.

Aufgabe PROJ-7-2

[Module **`server`**, **`client`**] Kommandozeilenoptionen für Server und Client

Ermöglichen Sie Kommandozeilenoptionen bei Server und Client. Diese können beim Aufruf der Programme angegeben werden, um Werte aus der Konfigurationsdatei (bzw. Standardwerte) direkt überschreiben zu können. Beim Server soll mit der Option `-c` und einem Argument eine alternative Konfigurationsdatei angegeben werden können, beim Client zusätzlich dazu noch eine IP-Adresse, zu der sich der Client verbinden soll (Option `-i`) sowie den Port (Option `-p`). Ein Aufruf des CLients könnte dann so aussehen:
`./client -i 127.0.0.1 -c client.conf -p 4444`

Aufgabe PROJ-7-3

[Module **`server`**, **`util`**] Polling und Signale im Server

Hinweis:
Sie haben bisher wahrscheinlich nur asynchrone Signalkonzepte kennengelernt. Dabei wird ein Signalhandler installiert, der ausgeführt wird, sobald ein Signal eintrifft. Dazu wird der normale Fluss des Programms unterbrochen. Dadurch hat der Signalhandler nur Zugriff auf globale Variablen. Außerdem sind nicht alle Bibliotheksfunktionen dazu geeignet in Signalhandlern benutzt zu werden (eine Liste dieser Funktionen finden Sie in der Manpage der Sektion 7 zu `signal`).
Deshalb wurde mit dem Linux-Kernel 2.6.22 eine neues Konzept eingeführt: `signalfd()`. Durch diese Funktion wird ein Filedeskriptor angelegt, an dem beim Eingang eines Signals eine Struktur mit Informationen über das Signal lesbar wird. Der Filedeskriptor kann auch in einem `poll()`-Aufruf verwendet werden. Durch dieses Konzept können Signale synchron in den Hauptschleifen der Programme bearbeitet werden, das Programm muss nicht mehr extra unterbrochen werden.
Dabei gibt es aber trotzdem einige Dinge die beachtet werden sollten. Da die Signale nicht mehr den Programmablauf unterbrechen (durch das Blockieren der Signale werden auch Systemaufrufe wie `read()` oder `write()` nicht mehr unterbrochen), sollte man darauf achten, sie möglichst zeitnah bearbeiten zu können. Für das Projekt heißt das, dass möglichst nur in einem `poll()`-Aufruf blockiert werden sollte, in dem auch der Signal-Filedeskriptor überprüft wird. Daher empfiehlt sich der Einsatz von nicht-blockierenden Filedeskriptoren. Auch werden die Filedeskriptoren bei einem `fork()` nicht geschlossen - das heißt, ein Kind könnte die Signale des Vaterprozesses abfangen. Schließlich benötigt der `signalfd()`-Aufruf mindestens Kernel 2.6.22 und glibc 2.8. Sollte Ihr System diese Voraussetzungen nicht erfüllen, so können Sie die Pseudoimplementierung verwenden, die sich in den beiden Ihnen auf der Website zur Verfügung gestellten Dateien befindet (`signalfd.c` und `signalfd.h` im Modul `util`). Um die Pseudoimplementierung zu verwenden, entkommentieren Sie die Zeile `#define _SIGFD_NOSUPPORT` in `signalfd.h`. Falls Ihr Kernel zwar ausreichend aktuell ist, Ihre glibc aber unterhalb von 2.8, so existiert keine `sys/signalfd.h` auf Ihrem System. In diesem Fall können Sie in der von uns zur Verfügung gestellten `signalfd.h` die Zeile `#define _SIGFD_KERNELSUPPORT` aktivieren, um zusammen mit `signalfd.c` die Funktionalität in Ihrem Kernel zu nutzen.

Um im Server Signale behandeln zu können, gehen Sie wie folgt vor:

a. Lesen Sie in der man-Page zu `signalfd()` nach, wie man einen Signal-Filedeskriptor einsetzt (siehe auch den Hinweis unten). Achten sie speziell darauf, dass Signale, die über einen Signal-Filedeskriptor empfangen werden sollen, mit `sigprocmask()` geblockt werden. Falls Sie keine solche man-Page haben, können Sie die aktuelle man-Page im Internet unter der Adresse `http://www.kernel.org/doc/man-pages/online/pages/man2/signalfd.2.html` einsehen.

b. Bauen Sie die Dateien `signalfd.h` und `signalfd.c` in Ihr `util`-Modul ein. Die Funktion `getSigfd()` kümmert sich darum, dass die über den Filedeskriptor empfangenen Signale blockiert werden.
Beachten Sie, dass bei der Pseudo-Implementierung mittels Signalhandler für Kernel kleiner als 2.6.22 die Signale nicht blockiert werden können. Trotz des gesetzten Flags `SA_RESTART` werden einige Systemaufrufe, wie zum Beispiel `poll()` nicht neugestartet, sondern liefern den Fehler `EINTR` zurück. Dieser muss dann abgefangen und der Systemaufruf neugestartet werden. Eine Liste der Systemaufrufe, die nicht neugestartet werden, finden Sie unter `http://www.kernel.org/doc/man-pages/online/pages/man7/signal.7.html`.

c. Benutzen Sie nun im Server die Funktion `getSigfd()`, um einen Signal-Filedeskriptor für die Signale `SIGINT`, `SIGQUIT` und `SIGCHLD` zu erhalten. Weisen Sie diesen Filedeskriptor der Variable `sigfd` Ihrer `struct actionParameters` zu. Erweitern Sie dann die Server-Hauptschleife, analog zum Client, um einen `poll()`-Aufruf, in dem Sie die Filedeskriptoren für Signale und vom Client eingehende Daten auf `POLLIN` überprüfen. Wie oben beschrieben können sie in diesem Fall eine Struktur `struct signalfd_siginfo`, aus dem `signalfd`-Filedeskriptor lesen und so genauere Informationen über das Signal erfahren.

d. Beenden Sie die Hauptschleife und das Programm für die Signale `SIGINT` und `SIGQUIT` sauber. Im Fall von `SIGCHLD` loggen Sie bitte die PID und den Rückgabewert des beendeten Kindes. Das Signal `SIGINT` können Sie durch drücken der Tastenkombination `STRG-C` auslösen.

Aufgabe PROJ-7-4

[Module **`server`**] Comfork

Damit der Server mehrere Clients gleichzeitig bearbeiten kann, soll nach der Verbindung eines Clients, die über `accept()` eingegangen ist, ein neuer Prozess abgespalten werden. Dieser Prozess kümmert sich dann um alle Anfragen des Clients. Gehen Sie dazu folgendermaßen vor:

a. Lagern Sie die bisherige Hauptschleife, die die Kommandos des Clients bearbeitet, in eine eigene Funktion `int comfork(struct actionParameters *ap, union additionalActionParameters *aap);` aus. Dies ist jetzt die Hauptschleife, die für jeden verbundenen Client in einem eigenen Prozess abläuft. Diese Funktion soll den Rückgabewert von `processIncomingData()` zurückgeben, oder -1 im Fehlerfall. Die Rückgabewerte sind also dieselben wie die einer Aktion aus dem `protocol`-Modul.

b. Erstellen Sie an der Stelle des `accept()`-Aufrufs in Ihrer `main()`-Funktion eine neue Hauptschleife, um neue Verbindungen zu akzeptieren. Verwenden Sie auch hier einen `poll()`-Mechanismus für den passiven Socket (anstelle einer schon geöffneten Verbindung) und den Signal-Filedeskriptor.

c. Zeigt `poll()` ein `POLLIN` auf dem passiven Socket an, dann akzeptieren Sie die Verbindung mittels `accept()`, führen Sie einen `fork()` durch und betreten Sie im Kind die Funktion `comfork()`. Achten Sie darauf, den `comfork`-Prozess und mögliche weitere Kindprozesse zu beenden und nicht weiter die Hauptschleife bearbeiten zu lassen. Beachten Sie dazu besonders die Rückgabewerte -2 und -3 aus dem `protocol`-Modul. Sobald Sie die Verbindung durch `accept()` angenommen haben, sollten Sie den passiven Socket im Kind schließen.

d. Verwenden Sie in beiden Schleifen denselben Signalbehandlungsteil. Nach einem `fork()`, also bevor Sie die Funktion `comfork()` betreten, müssen Sie einen neuen Signal-Filedeskriptor mit der Funktion `getSigfd()` anfordern, da Sie sonst die Signale des Vaters abfangen würden.

Aufgabe PROJ-7-5

[Module **`server`**] Shared Memory für die Filelist

Erzeugen Sie im Server ein Shared Memory für die Dateiliste des Servers. Die Dateiliste enthält Namen und Größe aller im Netz verfügbaren Dateien und ihren Ort (identifiziert durch IP und Port). Weisen Sie die ID, die Sie durch `shmget()` erhalten, dem Feld `shmid_filelist` der Struktur `struct serverActionParameters` Ihres Servers zu. Verwenden Sie dann die Funktion `initArray()` aus dem `util`-Modul, um das Shared Memory an das Feld `struct array *filelist` derselben Struktur zu binden und zu initialisieren. Verwenden Sie als Größe eines Eintrages (`itemsize`) `sizeof(struct flEntry)`. Die Struktur `struct flEntry` hat folgende Definition, fügen Sie diese in Ihre `util.h`-Datei ein:

```
/* struct flEntry
 * Represents a file in the network.
*/
struct flEntry {
        struct in_addr ip;
        uint16_t port;
        char filename[FILENAME_MAX];
        unsigned long size;
};
```

Aufgabe PROJ-7-6

[Module **`client`, `protocol`**]Client-Protokoll

Im Client werden Serverantworten bisher nur an der Konsole ausgegeben. Dies soll nun erweitert werden. Wie Sie an der Funktion `reply()` aus dem `protocol`-Modul erkennen können, sendet der Server vor seinem Antworttext immer einen dreistelligen Zahlencode. Die Konstanten dafür sind in `protocol.h` definiert. Im Falle von `REP_COMMAND`, mit dem Wert 300, fordert der Server Aktionen vom Client an, die keine Benutzerinteraktion erfordern. Dies geschieht in 2 Fällen: Der Server hat einen neuen passiven Socket angelegt, um die Dateiliste zu empfangen und fordert den Client auf, sich zu diesem zu verbinden oder es liegen Suchergebnisse nach einer Client-Anfrage vor, die sich der Client ebenfalls bei einem vom Server angelegten Socket abholen kann.

Um diese Funktionalität im Client zu ermöglichen, wird ein neues Protokoll – das Client-Protokoll – eingesetzt. Implementieren Sie dieses Protokoll. Sie können sich dabei am schon vorhandenen Server-Protokoll und dem folgenden Vorgehen orientieren:

a. Legen Sie im Modul `protocol` die Dateien `client_protocol.c` und `client_protocol.h` für das neue Protokoll an.

b. Fügen Sie die folgenden drei (noch leeren) Funktionen ein:

   - `int client_resultAction(struct actionParameters *ap, union additionalActionParameters *aap);`
   - `int client_sendlistAction(struct actionParameters *ap, union additionalActionParameters *aap);`
   - `int client_unknownCommandAction(struct actionParameters *ap, union additionalActionParameters *aap);`

c. Legen Sie außerdem eine Protokollstruktur mit den beiden ersten Aktionen an. `client_unknownCommandAction()` ist die `defaultAction`.

d. Inititalisieren Sie im Client eine Struktur `actionParameters` und eine Struktur `clientActionParameters` mit so vielen Feldern wie möglich. Weisen Sie dem Feld `prot` der `struct actionParameters` das neue Protokoll und einer `union additionalActionParameters` die `struct clientActionParameters` zu.

e. Lagern Sie die Funktionalität beim Eingang einer Servernachricht in eine neue Funktion `int processServerReply(struct actionParameters *ap, union`

`additionalActionParameters *aap);` aus. Diese Funktion soll aufgerufen werden, wenn `poll()` in der Hauptschleife eingehende Daten vom Server meldet. In der Funktion wird dann genau einmal mittels `readToBuf()` gelesen und mit dem Tokenizer versucht eine oder mehrere neue Zeilen zu finden. Ist eine neue Zeile gefunden, so wird das erste Wort daraus in eine Zahl `code` umgewandelt. Sobald dies geglückt ist, soll die Funktion `int processCode()` (nächste Teilaufgabe) aufgerufen werden. Sie liefert auch den Rückgabewert, den `processServerReply()` zurückgeben soll. Eine Zeile vom Server könnte wie folgt aussehen:
`200 Operation completed successfully.\r\n`

f. Schreiben Sie eine weitere Funktion `int processCode(int code, struct actionParameters *ap, union additionalActionParameters *aap);`. Diese führt anhand des zuvor extrahierten Codes verschiedene Aktionen durch:

   (a) `REP_OK`: Ausgabe von OK auf der Konsole.

   (b) `REP_TEXT`: Ausgabe des Textes vom Server auf der Konsole.

   (c) `REP_COMMAND`: Anstossen des `client`-Protokolls durch Aufruf der `protocol`-Funktion `processCommand()`. Der Rückgabewert aus dieser Funktion soll auch durch `processCode()` zurückgegeben werden.

   (d) `REP_ERROR`: Ausgabe des Fehler-Textes vom Server auf der Konsole.

   (e) `REP_FATAL`: Ausgabe des Textes vom Server auf der Konsole und Rückgabe von -1, so dass die Client-Hauptschleife und damit die Verbindung zum Server beendet wird.

g. Zuletzt bearbeiten Sie die Funktion `client_unknownCommandAction()` so, dass die Verbindung beendet wird, da der Server in diesem Fall offensichtlich nicht dieselbe Sprache spricht wie der Client.

Aufgabe PROJ-7-7

[Module **`server`**, **`client`**, **`connection`**, **`protocol`**] Übertragung von Dateilisten

In dieser Aufgabe soll der Client seine Dateiliste an den Server übertragen. Dazu wird im Server-Protokoll eine neue Aktion `filelistAction()` und im `connection`-Modul die Funktion `recvFilelist()` zum Empfangen der Liste angelegt.

Gehen Sie wie folgt vor:

a. Legen Sie die beiden Funktionen an.

b. Legen Sie in `filelistAction()` einen neuen passiven Socket an. Schicken Sie daraufhin an den Client die Meldung `SENDLIST SOCKET %d` mit dem Code `REP_COMMAND` und dem Port des neuen Sockets an der Stelle von `%d`. Führen Sie einen `fork()` durch, in dem Sie auf eine eingehende Verbindung vom Client warten. Der Vaterprozess (`comfork()` der mit dem Client kommuniziert) kann jetzt den neuen Socket schließen und normal weiterarbeiten.

c. Im Client wird daraufhin die Funktion `client_sendlistAction()` aufgerufen. Verwenden Sie auch in dieser Funktion einen `fork()`-Aufruf, um ein Kind zu erzeugen, das die Dateiliste überträgt. Der Haupt-Client bleibt dadurch währenddessen fähig Benutzereingaben zu verarbeiten. Verbinden Sie sich im abgespaltenen Prozess zu dem im Server angelegten Socket. Dazu müssen Sie zuerst das eingegangenen Server-Kommando weiter zerlegen, um den Port des Sockets im Server zu erfahren. Sobald die Verbindung hergestellt ist, übertragen Sie die Dateiliste des Verzeichnisses `conf.share` mit der zuvor geschriebenen Funktion `parseDirToFd()` über den Socket. Danach können Sie den Socket schließen und kehren mit dem Rückgabewert -2 oder im Fehlerfall -3 zurück, so dass das Kind in der Client-Hauptschleife beendet wird.

d. Um die Dateiliste im Server zu empfangen, benötigen Sie noch die Funktion `int recvFileList(int sfd, struct actionParameters *ap, struct serverActionParameters *sap);`. Das Feld `sfd` bezeichnet den geöffneten Socket. Verwenden Sie in dieser Funktion `tokenizer`-Funktionen, um die eingehenden Daten zeilenweise zu zerlegen, füllen Sie diese korrekt in eine `struct`

`flEntry` und verwenden Sie die Funktion `addArrayItem()`, um die `flEntry`-Struktur zu dem Shared-Memory-Array `sap->filelist` hinzuzufügen. Vergessen Sie nicht, den Zugriff auf das Shared Memory per Semaphore zu schützen!

e. Bauen Sie nun schließlich die Funktion `recvFilelist()` in den im Server geschaffenen Kinprozess ein, und beenden Sie die Action-Funktion ebenfalls mit -2 oder -3, woraufhin der Kindprozess in der Hauptserverschleife beendet wird, nachdem alle Ressourcen freigegeben wurden.

Hinweis:
Die Felder `ip` und `port` werden nicht mitübertragen. Benutzen Sie die Funktion `getpeername()`, um die entsprechenden Felder der `serverActionParameters`-Struktur zu füllen, sobald sich ein neuer Client zum Server verbunden hat. Verwenden Sie diese Felder, um beim Hinzufügen der einzelnen neuen Dateieinträge in die Liste, die `flEntry`-Struktur zu vervollständigen.

Aufgabe PROJ-7-8

[Module **`tokenizer`**, **`server`**, **`client`**, **`connection`**] Suche

Auf der Praktikumswebsite wird Ihnen die Funktion `int searchString(char *string, ...);` zur Verfügung gestellt. Sie verwendet die Tokenizer-Funktion `searchToken()`, um eine einfache Suche zu realisieren. Sie wird auf den String `string` angewendet, wobei der zu suchende Begriff als Separator angegeben wird. Je nach Rückgabewert kann dann festgestellt werden, ob der Suchstring in dem zu durchsuchenden String enthalten war. Eine Besonderheit ist hier der Fall `ts != 0`, der eintritt, wenn der Suchstring am Anfang vorkommt.

Um die Suche zu realisieren gehen Sie wie folgt vor:

a. Erstellen Sie im Server-Protokoll eine neue Funktion `searchAction()`. Diese wird durch eine Eingabe von "search <suchbegriff>" im Client angestoßen.

b. Erzeugen Sie in der neuen `searchAction()` einen neuen passiven Socket (die Kommunikation läuft genauso wie bei der Übertragung der Dateiliste, nur in die andere Richtung), und senden Sie dem Client das Komando `RESULT SOCKET %d`, den Code `REP_COMMAND` und den Port des neuen Sockets anstelle von `%d`. Dies soll im Client die Funktion `client_resultAction()` anstoßen, die sich zu dem neuen Socket verbindet. Führen Sie danach im Server und im Client jeweils einen `fork()` durch, in denen das Senden und Empfangen der Resultate stattfinden soll.

c. Legen Sie im Client ein Shared Memory für die Resultate an, und erzeugen Sie darin ein Array von `flEntry`-Strukturen. Für das Shared Memory ist das Feld `results` der Struktur `clientActionParameters` vorgesehen.

d. Schreiben Sie im `connection`-Modul eine Funktion `int sendResult(int fd, struct actionParameters *ap, struct serverActionParameters *sap)`, die durch die Dateiliste des Servers iteriert und dort für jeden Dateinamen eine Suche nach dem Suchebegriff durchführt (dieser sollte sich im Feld `comword` der Struktur `actionParameters` befinden). Der Parameter `int fd` ist der verbundene Socket, in den zeilenweise der Inhalt der `flEntry`-Struktur der Datei geschrieben wird, bei der der Suchstring gefunden wurde.

e. Schreiben Sie ebenfalls im `connection`-Modul eine Funktion `int recvResult(int fd, struct actionParameters *ap, struct array *results)`, die die Daten der `sendResult`-Funktion aus dem Socket `fd` empfängt und in das Shared Memory Array `results` schreibt. Sie können sich dabei sehr nah an der `recvFilelist()`-Funktion orientieren, nur müssen sie jetzt noch die IP-Adresse und den Port mitempfangen.

f. Bauen Sie die beiden Funktionen `sendResult()` und `recvResult()` in die vorher geforkten Kindprozesse der `searchAction()` und `client_resultAction()` ein, so dass die Suchresultate im Client ankommen.

g. Geben Sie schließlich im Client über den `Consoler` die gefundenen Resultate aus. Geben Sie auch die Nummer des Array-Elements jedes Resultats mit aus.