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 5— Vertiefung & Projekt I: Logging, Parsergenerierung, Datenstrukturen, Datei- und Verzeichnisoperationen

Abgabedatum theor. Aufgaben	Abgabedatum prakt. Aufgaben	Deadline Projektaufgaben
—	—	19.12.

Projekt-Aufgaben (Blatt 5)

Hinweise

- Folgende Dateien werden auf der Praktikumswebseite zur Verfügung gestellt:
  - `./util/util.h`
  - `./util/writef.c`
  - `./logger/logger.h`
  - `./tokenizer/tokenizer.h`
  - `./tokenizer/tokenizer.c.rahmen`
  - `./config/server.conf`
  - `./config/config.h`
- Beachten Sie: Alle Schnittstellenbeschreibungen sind verbindlich! Nehmen Sie insbesondere keine Änderungen an vorgegebenen Funktionssignaturen vor!
- Im Projekt kommen mehrere Module vor. Ein Modul ist dabei immer einfach ein Unterverzeichnis in ihrem durch SVN versionierten Projektverzeichnis. Normalerweise enthält ein Modul X je eine gleichnamige X.c und X.h Datei, sowie ein Makefile, es können aber auch mehrere .c und .h Dateien vorkommen. Wenn eine Aufgabe ein neues Modul erfordert, legen Sie das Verzeichnis und die dazugehörigen Dateien bitte an, auch wenn dies nicht explizit in der Aufgabenstellung gefordert wird.
- Generell gilt: Schreiben Sie zu jedem neuen Modul ein Makefile, das die Übersetzung des Moduls (`gcc -c ...`) durch den Aufruf von `make` ermöglicht. Außerdem soll das Makefile ein Ziel `clean` enthalten, so dass beim Aufruf von `make clean` alle zuvor erzeugten (Objekt-)Dateien wieder entfernt werden (die Quelldateien sollen natürlich erhalten bleiben!).
- Achten Sie darauf, dass alle mit der Zeit neu hinzukommenden Dateien durch SVN versioniert und mitverwaltet werden.

[0]Stand: 6. Dezember 2010

Aufgabe PROJ-5-1

Legen Sie ein Subversion-Projekt `sysprak` an. Sie können dafür zum Beispiel das Home-Verzeichnis ihres CIP-Pool-Accounts verwenden. Stellen Sie sicher, dass alle Gruppenmitglieder sowie Ihr Tutor Lese- und Schreibzugriff auf das Repository haben.

Legen Sie als Gruppe Commit-Richtlinien fest, und beschreiben Sie diese in einer Textdatei (`commitrichtlinien.txt`) als Teil des Projekts. Solche Richtlinien umfassen zum Beispiel Abmachungen über die Häufigkeit von Updates, Inhalte der Log-Nachrichten, ob im Repository immer eine lauffähige Version vorliegen muss, usw.

Hinweise

- Das (fertige) Projekt wird am Ende aus verschiedenen Modulen bestehen. Jedes davon wird durch ein eigenes Unterverzeichnis repräsentiert. Für die auf diesem Blatt betrachteten Module legen Sie im Projektverzeichnis bitte folgende Unterordner an: `util`, `server`, `config`, `directoryParser`, `tokenizer` und `logger`. Alle zu einem Modul gehörenden Code-Dateien sollen sich in den jeweiligen Unterordnern befinden.
- Zum Umgang mit SVN hat die Rechnerbetriebsgruppe eine Anleitung erstellt, die unter `http://www.rz.ifi.lmu.de/Dienste/Subversion` zu finden ist.

Aufgabe PROJ-5-2

[Modul **`util`**] Funktionen für Pufferspeicher
Erstellen Sie die Datei `util.c`. Implementieren Sie darin die Funktionen `createBuf()`, `copyBuf()`, `flushBuf()` und `freeBuf()` gemäß der Schnittstellenbeschreibungen in der vorgegebenen Headerdatei `util.h`.

Alle Funktionen verwenden die Struktur `struct buffer`, die einen Pufferspeicher (`char *buf`) mit der Gesamtgröße (`unsigned int bufmax`) und seinem momentanen Füllstand (`unsigned int buflen`) beschreibt.

Aufgabe PROJ-5-3

[Modul **`util`**] Ein- und Ausgabe für Pufferspeicher
Fügen Sie die drei folgenden Funktionen `writeWrapper()`, `readToBuf()` und `writeBuf()` in die schon bestehenden `util.c`- und `util.h`-Dateien ein.

a. `writeWrapper()`

   Schreiben Sie eine Funktion, die gesichertes Schreiben in einen Filedeskriptor gewährleistet, mit folgender Spezifikation (analog zur Bibliothekfunktion `write()`):

   ```
   ssize_t writeWrapper (int fd, const void *buf, size_t count);
   ```

   Verwenden Sie sich hierfür die Funktion `write()`. Der Rückgabewert der Funktion soll, analog zur `write()`-Funktion, die Anzahl der tatsächlich geschriebenen Bytes oder -1 im Fehlerfall sein.

b. `writef()`

   Die Funktion `writef()` (siehe Datei writef.c) benutzt die von Ihnen geschriebene Funktion `writeWrapper()`, um das Schreiben in einen Filedeskriptor zu ermöglichen. Dabei wird, wie in der Bibliotheksfunktion `printf()`, eine variable Argumentenliste mit einem Formatstring unterstützt. Um aus der variablen Argumentenliste den endgültig zu schreibenden String zu extrahieren, wird eine weitere Funktion `vStringBuilder()` eingesetzt (vgl. `vprintf()`). Sie gibt einen neu reservierten Speicherbereich (vgl. `malloc()`) mit dem zusammengesetzen String zurück. Dieser temporäre Speicherbereich muss danach wieder freigegeben werden (`free()`).

c. `readToBuf()`

Schreiben Sie eine Funktion, die möglichst viele Bytes aus einem Filedeskriptor liest und in einen Pufferspeicher schreibt. Benutzen Sie folgende Spezifikation:

```
ssize_t readToBuf (int fd, struct buffer *buf);
```

Verwenden sie die Funktion `read()`. Der Rückgabewert ist, analog zur Funktion `read()`, die Anzahl der gelesenen Bytes oder -1 im Fehlerfall.

d. `writeBuf()`

Schreiben Sie eine Funktion, die alle Bytes aus einem Puffer in einen Filedeskriptor schreibt. Verwenden Sie die folgende Spezifikation:

```
ssize_t writeBuf (int fd, struct buffer *buf);
```

Verwenden Sie zum Schreiben des Puffers die Funktion `writeWrapper()`. Der Puffer soll durch das Schreiben geleert werden. Achten Sie entsprechend darauf, dass alle Werte in der Pufferspeicherstruktur am Ende korrekt gesetzt sind. Die Rückgabewerte der Funktion sind dieselben, wie die der Funktion `writeWrapper()`.

Hinweise

- `writeWrapper()`
  Diese Funktion ist nötig, da die Bibliotheksfunktion write nicht garantiert, dass die Anzahl an Bytes, die durch `size_t count` vorgegeben werden, auch wirklich geschrieben werden. Unterbrechungen können z.B. durch Signale auftreten. Da später auch nichtblockierende Filedeskriptoren verwendet werden und Unterbrechungen durch Signale auftreten könnten, sollen die Fehler `EAGAIN` und `EINTR` ignoriert werden (d.h. falls diese Fehler auftreten soll trotzdem versucht werden weiterzuschreiben).

  Tipp:
  Machen Sie sich in der man-Page mit der Bedeutung des Rückgabewerts von write() vertraut und lassen Sie die Funktion anhand dieses Wertes weiterarbeiten. Für die Abfrage der errno-Variablen benötigen Sie die Header-Datei `errno.h`.

- `writef()`
  Da die Funktion `vStringBuilder()` das Makro `va_copy` einsetzt und dieses erst mit dem ISO-Standard C99 eingeführt wurde, können Sie das Programm nicht mit der Option `-ansi` kompilieren (für Details siehe `man gcc`). Beide Funktionen werden hier vollständig vorgegeben, das heißt, Sie müssen sie nur in Ihr Projekt einbauen. Sehen Sie sich jedoch bitte die man-Page zu `vprintf()` an, insbesondere die Funktion `vStringBuilder` stammt größtenteils aus dieser man-Page.

- `readToBuf()`
  Da auch nicht-blockierende Filedeskriptoren verwendet werden und Unterbrechungen von Signalen korrekt bearbeitet werden sollen, müssen die Fehlerfälle von `read()`, in denen `errno` den Wert `EAGAIN` oder `EINTR` annimmt, extra behandelt werden. In diesem Fall soll die Funktion -2 zurückgeben. Die Definitionen für `errno`, `EAGAIN` und `EINTR` finden sich in der Headerdatei `errno.h`

- Für die Typen `size_t` und `ssize_t` müssen Sie die Header-Datei `sys/types.h` inkludieren.

Aufgabe PROJ-5-4

[Modul **`server`**] - das Server-Hauptprogramm

Erstellen Sie die Datei `server.h` und `server.c` mit einer `main()`-Funktion und legen Sie neben dem normalen Modul-Makefile auch im Hauptmakefile (im Projektverzeichnis) ein Target für das Serverprogramm an, welches die Module `util` und `server` zu einem Programm verbindet.

Testen Sie in der `main()`-Funktion die Puffer- und Ausgabe-Funktionen aus dem `util`-Modul ausgiebig! Aufgrund der häufigen Verwendung in vielen Modulen können Fehler in den Implementierungen der Ausgabe-Funktionen später zu unerwünschten und schwer zu diagnostizierenden Nebeneffekten führen. Verwenden Sie dieses Programm auch zum Testen der folgenden Aufgaben und fügen Sie die erstellten Module im Makefile hinzu.

Tipp:
Sie können als Filedeskriptor für die Ausgabe auf der Konsole auch den Wert `STDOUT_FILENO` aus der Header-Datei `unistd.h` benutzen.

Aufgabe PROJ-5-5

[Modul **`logger`**]
Das Modul `logger` übernimmt das Schreiben von Log-Meldungen in eine Datei. Anhand dieser Log-Meldungen sollen der Programmablauf und etwaige aufgetretene Probleme nachvollzogen werden können. Da mit mehreren Prozessen gearbeitet wird, aus denen gleichzeitig geloggt werden soll, wird der `logger` so implementiert, dass er als eigener Prozess die Leseseite einer Pipe ausliest, in die alle anderen Prozesse von der Schreibseite aus Logmeldungen absetzen können. Die Funktion `logger` übernimmt dieses Auslesen der Pipe und schreibt die Meldungen in eine Datei. Sobald alle Prozesse, die Log-Meldungen schreiben wollen, beendet sind und damit ihre Seiten der Pipe geschlossen haben, soll auch der `logger`-Prozess terminieren.

Log-Meldungen sollen standardmäßig mit einem Zeitstempel und der PID des loggenden Prozesses versehen werden. Damit es nicht zu zerstückelten Logmeldungen kommen kann, muss die Pipe mit einer Semaphore geschützt werden, sobald mehrere Prozesse gleichzeitig loggen wollen (siehe später). Dies übernimmt die Funktion `logmsg`.

a. `logger()`

   Schreiben Sie eine Funktion, die aus einer Pipe liest und das Gelesene in eine Datei schreibt, bis alle Schreibseiten der Pipe geschlossen wurden oder ein Fehler auftritt. Die Pipe wie auch die Datei sollen (schon geöffnet) als Filedeskriptoren an die Funktion übergeben werden. Die Spezifikation sieht wie folgt aus:

   ```
   int logger(int pipefd, int filefd);
   ```

   Verwenden Sie hierfür einen Puffer und die Funktionen `readToBuf()` und `writeBuf()` aus dem Modul `util`.
   Rückgabewert: -1 im Fehlerfall, 1 sonst

   Tipp:
   Lesen Sie in der man-Page der Funktion `read()` nach, wie festzustellen ist, dass alle Schreibseiten einer Pipe geschlossen sind (bis auf wenige Fehlerfälle gibt die Funktion `readToBuf()` dieselben Werte wie `read` zurück).

b. `logmsg()`

   Schreiben Sie eine Funktion, die ähnlich wie die Funktion `printf()` einen Formatstring (Argument `const char *fmt`) und beliebig viele Argumente verarbeiten kann. Die daraus erzeugte Nachricht soll in eine Pipe (Filedeskriptor, Argument `pipefd`) geschrieben werden. Zusätzlich soll ein `loglevel` übergeben werden, anhand dessen die Nachrichten gefiltert werden können (durch bitweisen Abgleich; vordefinierte Werte für loglevel finden Sie in der Datei logger.h, die auf der Praktikumswebseite zur Verfügung gestellt wird). Die Funktion `logmsg()` soll folgende Signatur aufweisen:

   ```
   int logmsg(int semid, int pipefd, int loglevel, const char *fmt, ...);
   ```

   Das erste Argument `semid` können Sie noch unbeachtet lassen, dies dient später dem Einsatz einer Semaphore. Ein Code-Beispiel finden Sie in den begleitenden Hinweisen zu diesem Aufgabenblatt.

Hinweise

- Es soll ein Zeitstempel sowie die PID des loggenden Prozesses vor die eigentliche Logmeldung gehängt werden. Verwenden Sie die Funktionen `time()` und `localtime()` um die lokale Zeit zu bekom-

men und dann die Funktion `strftime()` um diese in einen lesbaren String umzuwandeln. Die PID des Prozesses können Sie mit der Funktion `getpid()` abfragen.

- Rückgabewert soll 1 bei Erfolg, -1 bei Fehler sein.

Aufgabe PROJ-5-6

[Modul **`tokenizer`**]
Der Tokenizer ist ein zentraler Teil des Projekts. Er wird verwendet, um Textdaten zu zerlegen. Dabei kommt er in vielen verschiedenen Variationen zum Einsatz, unter anderem zum parsen der Konfigurationsdatei, zum Zerteilen der Kommandos von Server und Client und zur Suche nach Dateien. Seien sie also bei der Implementierung besonders sorgfältig und testen Sie den Tokenizer ausgiebig.

Legen Sie das Modul `tokenizer` mit inkl. seiner Header- und C-Dateien sowie einem Makefile an und implementieren Sie folgende Funktionen:

a. Schreiben Sie eine Funktion, die in einem Puffer `struct buffer *buf` versucht ein Token zu finden. Die einzelnen Tokens sind dabei durch einen Separator voneinander getrennt. Separatoren direkt am Anfang des Puffers sollen automatisch entfernt werden. Die Funktion hat folgende Signatur:

```
int searchToken (int *tokenstart, int *tokenlength, int *fulllength,
struct buffer *buf, char *separator);
```

Sie füllt die per Zeiger übergebenen Integer-Werte wie folgt:

- `tokenstart` ist der Anfang des gefundenen Tokens und im Normalfall 0, wenn keine Separatoren am Beginn des Buffers gefunden wurden.
- `tokenlength` ist die Länge des Tokens ohne den darauffolgenden Separator.
- `fulllength` schließlich ist die komplette Länge des bearbeiteten Puffer-Teils inklusive der Länge der Separatoren am Anfang, der eigentlichen Tokenlänge und der Länge des Separators

Wird ein Separator gefunden, so ist der Rückgabewert 1. Falls kein Separator gefunden wird ist der Rückgabewert 0. `tokenstart` soll trotzdem mit dem richtigen Wert belegt sein, `tokenlength` soll die Länge des restlichen Puffers enthalten und `fulllength` die komplette Pufferlänge.

b. Erweitern Sie die Funktion `searchToken()` so, dass sie auch mehrere Separatoren als Eigabe bearbeiten kann. Ändern Sie hierzu das Argument `char *separator` in `va_list ap` und verwenden sie das Makro `va_arg`, um die variable Argumentenliste `va_list ap` abzuarbeiten. Beachten Sie, dass um jeden Aufruf der Funktion `tokenizer()` die Makros `va_start` und `va_end` geschachtelt werden müssen und die Funktion, in der der Aufruf erfolgt, über eine Ellipse verfügen muss.

Hinweis:
Lesen Sie die man-Pages zu den Makros.

Die endgültige Version von `searchString()` trägt jetzt entsprechend folgende Signatur:
```
int searchToken (int *tokenstart, int *tokenlength, int *fulllength,
struct buffer *buf, va_list ap);
```

c. Schreiben Sie nun eine Funktion, die die Werte von `searchString()` dazu verwendet, ein Token aus einem durchsuchten Puffer zu extrahieren. Die Funktion soll die folgende Signatur haben:

```
int extractToken(struct buffer *buf, struct buffer *token, int
tokenstart, int tokenlength, int fulllength);
```

Der Rückgabewert ist 1 für Erfolg und -1 im Fehlerfall.

Tipp:
Verwenden Sie die Funktion `memmove()`, um das Token aus dem Puffer zu entfernen.

Aufgabe PROJ-5-7

[Modul **directoryParser**]

Legen Sie, wie auch bei den vorigen Aufgaben, das Modul `directoryParser` als Verzeichnis in ihrem SVN-Repository mit den Dateien `Makefile`, `directoryParser.c` und `directoryParser.h` an. Schreiben Sie nun eine Funktion mit der Signatur:

```
int parseDirToFD(int fd, const char * basedir, const char * subdir)
```

Diese Funktion soll mittels der Funktionen `opendir()` und `readdir()` (lesen Sie bitte die man-Pages dazu) das Verzeichnis `basedir/subdir` durchwandern und für jeden gefunden Eintrag die Funktion `stat()` ausführen, um weitere Informationen zu erhalten. Machen Sie sich auf der man-Page von `stat()` mit den Makros `S_ISREG` und `S_ISDIR` vertraut. Je nach Typ des Eintrags sollen folgende Aktionen ausgeführt werden:

- reguläre Datei: Der Dateiname (inklusive relativem Pfad aus dem Parameter `subdir`) wird mittels der Funktion `writef()` gefolgt von einem Zeilenumbruch (`\n`), gefolgt von der Größe der Datei und schließlich mit einem weiteren Zeilenumbruch (`\n`) abgeschlossen in `fd` geschrieben.
- Verzeichnis: Die Funktion soll sich rekursiv mit gleichem `basedir` aufrufen, allerdings soll an `subdir` das neue zu bearbeitende Verzeichnis angehängt werden.
- mit einem Punkt beginnende Datei: solche Dateien enthalten unter Unix oft sensible Information. Deswegen sollen alle Dateinamen, die mit einem Punkt beginnen, einfach übersprungen werden.
- alle anderen Typen (z.B. Named Pipes, Sockets, Geräte, etc): diese sollen wie die mit einem Punkt beginnenden Dateien ignoriert werden.

Der Rückgabewert der Funktion soll im Fehlerfall -1 sein, bei Erfolg 1.

Tipp:
Benutzen Sie zum Testen Ihrer Funktion einfach `STDOUT_FILENO` für den Parameter `fd`.

Aufgabe PROJ-5-8

[Modul **util**] Array-Funktionen

An einigen Stellen müssen mehrere Strukturen im Speicher ablegen werden (z.B. Einträge in die Dateiliste des Servers, Einträge in die Liste der verbundenen Clients oder Informationen über die Kindprozesse des Clients). Einige dieser Daten müssen in Shared Memories abgelegt werden, da sie von mehreren Prozessen zugegriffen werden sollen. Daher bietet sich eine Verwaltung als Array an, da Shared Memories nur einen Block-Speicher zur Verfügung stellen (eine Linked List, bei der für jedes Element einzeln Speicher reserviert würde, wäre schwierig zu implementieren). Um eine relativ einfach und einheitliche Datenstruktur zu erhalten, sollen die folgenden Funktionen für normalen Speicher wie auch für Shared Memories funktionieren.

Für ein Array soll die folgende Struktur verwendet werden:

```
struct array {
        size_t memsize;
        size_t itemsize;
        unsigned long itemcount;
        void *mem;
        int shmid;
};
```

Fügen Sie diese Struktur in ihre `util.h`-Datei ein. Die einzelnen Felder der Struktur bedeuten:

- `size_t memsize`: Die Größe des reservierten Speicherbereiches auf den `mem` zeigt.

- `size_t itemsize`: Die Größe eines Array-Elements.
- `unsigned long itemcount`: Die Anzahl der sich im Array befindenden Elemente.
- `void *mem`: Zeiger auf den für das Array reservierten Speicherbereich.
- `int shmid`: ID des Shared Memory Segments. Soll normaler Speicher verwendet werden, ist sie -1.

Erstellen Sie nun die folgenden Funktionen im Modul `util`:

a. `struct array *initArray(size_t itemsize, size_t initial_size, int shmid)`
Diese Funktion erzeugt ein neues Array. `itemsize` gibt die Größe eines Elements an, `initital_size` die anfänglich zu reservierende Größe bzw. die Größe des Shared Memories und `shmid` ist -1 für normalen Speicher oder die ID des Shared Memory mit der zuvor angegebenen Größe. Reservieren Sie den Speicher mittels `malloc()` oder fragen Sie die Adresse mittels `shmat` ab (d.h. `shmget()` muß schon vom Aufrufer ausgeführt worden sein). Achten Sie darauf, daß sich die beschreibende Struktur des Arrays und die Arraydaten selbst in demselben Speicherbereich befinden, d.h. `mem` zeigt auf das erste Byte hinter der Arraystruktur. Beachten Sie dies auch bei der Berechnung der Speichergröße `memsize` auf die `mem` zeigt (die Speichergröße ist um eine `struct array` kürzer als `initial_size`).
Rückgabewert der Funktion ist ein Zeiger auf die neu resevierte Array-Struktur oder `NULL` im Fehlerfall.

b. `void freeArray(struct array *a)`
Diese Funktion gibt ein Array je nach Typ mittels `free()` oder `shmdt` wieder frei.

c. `struct array *addArrayItem (struct array *a, void *item)` Diese Funktion fügt ein neues Element in das Array ein. Im Falle von normalem Speicher soll, falls nicht genügend Speicher für ein weiteres Element verfügbar ist, mittels `realloc()` versucht werden, den Speicherbreich zu vergrößern (z.B. um 10 Einheiten). Im Falle von Shared Memory ist das nicht möglich. Das neue Element `void *item` soll ans Ende des Arrays gehängt werden (in dem Array soll es nie Lücken geben).
Bei Erfolg wird der Zeiger auf die möglicherweise neu adressierte Array-Struktur zurückgegeben, im Fehlerfall `NULL`. Achten Sie bei der Benutzung dieser Funktion darauf, dass Sie Ihren ursprünglichen Zeiger auf das Array nicht bei einem Fehler verlieren, da Sie sonst den Datenbereich nicht mehr freigeben können.

d. `int remArrayItem (struct array *a, unsigned long num)`
Diese Funktion löscht das Array-Item `num` aus dem Array. Dabei wird wie bei einem normalen Array die Zählung bei 0 begonnen. Um keine Lücken entstehen zu lassen, muss, falls `num` nicht das letzte Element ist, das letzte Element an die Stelle `num` kopiert werden und anschließend hinten gelöscht werden.
Rückabewert: 1 bei Erfolg, -1 bei Fehler (überprüfen Sie `num` auf Gültigkeit!).

e. `void *getArrayItem(struct array *a, unsigned long num)`
Diese Funktion liefert einen Zeiger auf das `num`-te Element zurück oder `NULL`, falls es kein solches Element gibt.

f. `void *iterateArray(struct array *a, unsigned long *i)`
Diese Funktion iteriert über ein Array. Ihr Rückgabewert ist das von `i` referenzierte Element, oder `NULL` falls `i` größer wird als der Füllstand des Arrays (`array->itemcount`). Als Seiteneffekt erhöht sie i um 1. Sie wird mit einem mit 0 initialisierten Zähler `unsigned long *i` aufgerufen (Sie müssen hier eine Variable verwenden) und kann in einer `while()`-Schleife benutzt werden, bis sie `NULL` zurückliefert.

Aufgabe PROJ-5-9

[Modul **`config`**] Auslesen der Konfiguration aus einer Datei

Im Modul `config` soll die Konfiguration, die in einer Konfigurationsstruktur liegt, mit Standardwerten gefüllt und aus einer Datei ausgelesen werden können. Wir verwenden der Einfachheit halber dieselbe Struktur für Server und Client, weil sich einige Konfigurationsfelder überschneiden. Als Konfigurationsstruktur soll folgende verwendet werden:

```
#include <inttypes.h> /* uintX_t */
#include <stdio.h> /* FILENAME_MAX */
```

```c
#include <netinet/in.h> /* struct in_addr */


/* struct config
 * configuration structure for _both_, server and client (merged). */
struct config {
    struct in_addr   ip;
    uint16_t         port;
    char             logfile[FILENAME_MAX];
    uint8_t          loglevel;
    char             share[FILENAME_MAX];
    uint32_t         shm_size;
    /* ... */
};
```

Erstellen Sie das Modul `config`, wie immer mit einer Code-, Header- und Makefile-Datei. Erstellen Sie die beiden folgenden Funktionen:

a. `void confDefaults(struct config *conf);`
   Diese Funktion soll die `struct config` mit Standardwerten füllen, so dass man nicht manuell alle Werte angeben muss. Dazu sollte als erstes diese Funktion aufgerufen werden, bevor die Standardwerte dann mit Werten aus der Konfigurationsdatei und schließlich von der Kommandozeile überschrieben werden.

b. `int parseConfig (int conffd, struct config *conf);`
   Diese Funktion erwartet einen geöffneten Filedeskriptor `int conffd`, aus dem sie Konfigurationswerte einliest und in die `struct config *conf` speichert. Die Konfigurationswerte sollen zeilenweise (`\n` oder `\r\n`) in der Konfigurationsdatei stehen und aus einem Schlüsselwort und darauf folgendem Wert bestehen, die mit Whitespaces (`\t` oder Leerzeichen) voneinander getrennt sind. Außerdem sollen Kommentare möglich sein, die hinter einer Raute oder einem Semikolon beginnen. Verwenden Sie die Funktion `int getTokenFromStream()`, um einzelne Zeilen aus der Datei zu lesen und dann die Funktion `int getTokenFromBuffer()`, um diese Zeilen in Wörter zu zerteilen. Verwenden Sie die Funktion `strcmp()`, um zuerst die Schlüsselwörter mit den gelesenen Wörtern zu vergleichen, und schreiben Sie das nächste Wort in das korrekte Strukturfeld.

Hinweis:
Es steht bereits eine beispielhafte Konfigurationsdatei `server.conf` für den Server bereit, mit der Sie Ihre Tests durchführen können.

Hinweise, Hilfestellungen und Implementierungshilfen zu Aufgabenblatt 4

Übersicht über das Projekt

Bei diesem Projekt handelt es sich um ein einfaches Filesharing-System. Clients können sich zu einem Server verbinden, der Listen der auf dem Client angebotenen Dateien anfordert und diese Liste speichert. Die Clients wiederum können Suchanfragen an den Server stellen. Dieser antwortet mit den gefundenen Dateien und den Informationen (IP und Port), welcher verbundene Client diese Dateien besitzt. Die Clients fungieren auch selbst als Server, denn Datei-Downloads werden direkt von Client zu Client ausgeführt.

1. **Struktur**
   Das Projekt setzt sich aus den folgenden 10 Modulen zusammen:

   - `server`
     Modul für das Serverprogramm, das Verbindungen von Clients annimmt, ihre Dateilisten verwaltet und die Suche von Dateien ermöglicht
   - `client`
     Modul für das Clientprogramm, das es dem Anwender emöglicht sich zum Server zu verbinden und Dateilisten sowie Dateien im Netz verfügbar macht. Mit diesem Modul können Dateien ebenfalls gesucht und heruntergeladen werden können, sowie kleinere Funktionen zur Überwachung von Up- und Downloads ausgegeben werden. Das Modul stellt die Hauptschnittstelle zwischen Server, User und anderen Clients bereit.
   - `logger`
     Ermöglicht allen Modulen Log-Nachrichten mit Zeitstempel in Dateien zu schreiben
   - `consoler`
     Kümmert sich um die Verwaltung der Console, so dass mehrere Prozesse dem Benutzer quasi-gleichzeitig Nachrichten senden können
   - `util`
     Stellt diverse Hilfsfunktionen für alle anderen Module zur Verfügung
   - `tokenizer`
     Sorgt dafür, dass Datenströme („Streams“) und andere Daten anhand von Separatoren zertleilt werden können, um zuerst Zeilen und, in einem weiteren Schritt, einzelne Tokens (Wörter) in Zeilen herauszufiltern
   - `directoryParser`
     Durchsucht ein Verzeichnis und seine Unterverzeichnisse nach Dateien
   - `connection`
     Basisfunktionen, die für die Kommunikation zwischen Client und Server, wie auch zwischen den Clients selbst, unabdingbar sind
   - `config`
     Liest mit Hilfe des Tokenizers eine Konfigurationsdatei aus
   - `protocol`
     Definiert die netzseitigen Schnittstellen der Programmteile, sowie die Reaktion auf eingehende Anfragen. Wir verwenden an drei Schnittstellen Protokolle: Anfragen von Client zu Server, Anfragen von Server zu Client, die keine Nutzerinteraktion beinhalten, und Benutzereingaben, die der Client verarbeitet.

2. **Aufgaben von Server und Client**
   Der Server übernimmt folgende Aufgaben:

   - Verwalten einer Dateiliste der Dateien aller verbundenen Clients
     Die Liste aller Dateien der verbundenen Clients enthält Dateinamen (inklusive relativem Pfad), Größe der Datei und Informationen welcher Client sie anbietet. Diese Liste wird in einem *Shared Memory*

gehalten, da das Empfangen neuer Dateiinformationen und die Suche nach Dateien in jeweils eigenen Prozessen ablaufen soll.

- Suche nach Dateien
  Stellt ein Client eine Suchanfrage, so wird von seinem Server (der direkte Partnerprozess des Clients ist ein `comfork`-Prozess, der vom Hauptserver beim Eingehen einer neuen Verbindung erstellt wird) ein Such-Prozess abgespalten, der die Dateiliste nach dem Suchbegriff durchsucht und die potentiellen Ergebnisse dem anfragenden Client zurücksendet.

Der Client stellt folgende Funktionalität zur Verfügung, nachdem er die Verbindung zu einem Server aufgebaut hat:

- Eingabe von Benutzerkommandos
  Benutzerkommandos, die über den Consoler eingehen, werden entweder vom Client selbst bearbeitet oder an den Server weitergeschickt.
- Parsen eines Vezeichnisses
  Jeder Client macht ein Verzeichnis (und dessen Unterverzeichnisse) dem Netz verfügbar. Der `directoryParser` durchläuft dieses Verzeichnis und sendet zu Beginn der Verbindung und nach manueller Benutzeraufforderung die dabei erstellte Dateiliste an den Server.
- Resultate einer Suche
  Der Client hält eine Liste (Dateiename, Dateigröße, IP, Port) mit den Resultaten seiner letzten Suchanfrage. Nur Dateien aus dieser Liste können runtergeladen werden.
- Download (Fork)
  Zum runterladen einer Datei verbindet sich der Client zu einem entfernten Client und fordert dort die Datei an (durch Senden des Dateinamens). Dieselbe Verbindung wird von dem entfernten Client verwendet, um die Datei zu schicken.
- Upload (Fork)
  Ein *Listening-Socket* jedes Clients wird nur dazu verwendet, Verbindungen von anderen Clients anzunehmen, die einen Upload anfordern. Zuerst wird der Dateiname ausgelesen und diese Datei dann über dieselbe Verbindung zurück zum anfordernden Client übertragen. Das bedeutet auch, dass zum Download einer Datei keine Verbindung von außen auf einen Client nötig ist - dazu werden nur Verbindungen benötigt, die vom Client selbst initiiert werden.
- Bearbeiten von Serverkommandos
  Einige Kommandos (z.B. Dateiliste senden und Resultate empfangen) erfordern eine weitere Verbindung zum Server (siehe auch folgenden Abschnitt „Designkriterien“). Da diese nicht vom Server initiiert werden soll (der Client soll weitestmöglich auch hinter einer Firewall funktionieren), muss der Server über die Kommandoverbindung den Client dazu auffordern können, eine neue Verbindung aufzubauen.

3. **Design-Kriterien**
   Bei der Umsetzung des Projektes soll auf folgende Designkriterien geachtet werden:

   - Kommandos zwischen Client und Server
     Die Hauptverbindung zwischen Client und Server dient ausschließlich als Kontrollverbindung. Abgesehen von Kommandos (lesbarer Text) sollen keine Nutzdaten (binär) übertragen werden. Ein Kommando ist dabei einfach eine Zeile Text, die einzelnen Bestandteile des Kommandos werden durch Leerzeichen und Tabs getrennt.
   - Nutzdatenübertragung
     Da eine Übertragung größerer Datenmengen einige Zeit in Anspruch nehmen kann, dabei aber der Client weiterhin Benutzereingaben annehmen können soll, soll jede Nutzdatenübertragung in einem eigenen Prozess (fork) und auf einer eigens für diese Übertragung geöffneten Verbindung stattfinden. Dazu zählen auch Dateilisten und Suchergebnisse.
   - Rückgabewerte von Funktionen
     Es soll versucht werden einheitliche Rückgabewerte von Funktionen und Prozessen zu verwenden. Allgemein steht -1 für Fehler (weitere negative Werte können für differenziertere Fehlerrückgabe verwendet werden), 1 für Erfolg. 0 steht für spezielle Rückgabewerte (z.B. kein Token mehr gefunden).

4. **Die Protokolle**
   Das Protokollmodul ist dazu da, die eingehenden Kommandos zu interpretieren (erst in Zeilen und dann in Wörter zu trennen) und darauf eine Funktion auszuführen, die auf das Kommando reagiert und antwortet. Dabei kommt eine `struct protocol` zum Einsatz, die neben dem Kommandostring einen Zeiger (Pointer) auf die auszuführende Funktion und eine kurze Beschreibung enthält. Stimmt das erste Wort des Kommandos mit einem in der Protokollstruktur abgelegten Kommandostring überein, wird die zugehörige Funktion ausgeführt. Es kommen an 3 Stellen Protokolle vor:

   - Server-Protokoll
     Dieses Protokoll dient zum parsen der Befehle, die vom Client an den Server gesendet werden. Es beinhaltet u.a. das Senden einer aktualisierten Dateiliste und das Suchen von Dateien.
   - Client-Protokoll
     Dieses Protokoll kommt zum Einsatz, wenn der Server eine Aktion vom Client fordert, die keine Nutzereingabe benötigt. Der Server sendet bei seinen Antworten (ähnlich dem http-Protokoll) als erstes Wort eine Nummer, die die Art der Antwort (z.B. OK, Fehler oder eben Aktion erforderlich) charakterisiert. Normalerweise wird der Nutzer nur über die Serverantwort informiert und evtl. der mitgesendete Text ausgegeben. Ist aber eine zusätzliche Aktion notwendig (wie eine neue Verbindung aufbauen und Suchergebnisse empfangen), wird sie über das Client-Protokoll behandelt.
   - Stdin-Protokoll
     Dieses Protokoll parst die Benutzereingaben, die der Benutzer über die Standardeingabe (`STDIN_FILENO`) und den `consoler` eingibt. Dabei werden nur Kommandos herausgefiltert, die keine Serverinteraktion beinhalten (z.B. Ausgabe der letzten Resultate). Alles andere wird einfach an den Server weitergeschickt. So muss der Client nicht wissen welches Protokoll genau der Server einsetzt und neue Funktionalitäten sind einfach zu implementieren. Natürlich sollen Client und Server allerdings unterschiedliche Kommandos benutzen!

5. **Wichtige Datenstrukturen**

   - Einfache Datenstrukturen:
     - `struct buffer`
       Diese Struktur stellt einen Buffer mit seiner maximalen Größe, momentanen Füllstand und einen Zeiger auf den Speicherbereich dar.
     - `struct array`
       Struktur um ein Array aus Strukturen in einem Speicherbereich oder einem Shared Memory zu verwalten.
     - `struct flEntry`
       Enthält alle nötigen Daten, um eine Datei im Netzwerk zu beschreiben. Dazu gehören ihr Name, ihre Größe, die IP-Adresse und der Port des Clients auf dem sie liegt.
     - `struct childProcess`
       Eine Struktur um einen Kindprozess mit seiner PID und seinem Typen zu beschreiben.
   - Größere Datenstrukturen, die die verwendeten Ressourcen zusammenfassen:
     - `struct actionParameters`
       Enthält alle Ressourcen die direkt für den Betrieb eines Protokolls nötig sind. Dazu gehören drei Buffer (einen Eingangsbuffer, einen für daraus extrahierte Zeilen und einen für Wörter), die Semaphoren-ID, den Filedeskriptor über den die Kommunikation läuft, den `signalfd()`-Filedeskriptor für Signale, den Filedeskriptor für die Logger-Pipe und einen Pointer auf die Protokoll-Struktur. Außerdem wird bitweise in der `usedres`-Variable gespeichert, welche Filedeskriptoren noch geöffnet und welcher Speicher noch nicht freigegeben ist.
     - `struct clientActionParameters`
       Diese Struktur enthält weitere Ressourcen, die der Client benutzt - und auch Aktionen seiner Protokolle. Zu den Ressourcen gehören: Filedeskriptor der Pipe zum Consoler (entspricht `STDOUT_FILENO`, eine Array-Struktur im Shared Memory für die Resultate, den Filedeskriptor der Verbindung zum Server und eine Array-Struktur für die Kindprozesse des Clients.

- `struct serverActionParameters`
  Enthält weitere Serverressourcen wie ein Shared Memory für die Dateiliste.
- `union additionalActionParameters`
  Eine Union aus serverActionParameters und clientActionParameters. Im Unterschied zu einer Struktur enthält eine Union jeweils nur eines ihrer Felder und hat genau die Größe des größten iherer Felder. Diese Union dient dazu, Actions mit einheitlichen Signaturen verwenden zu können.

Code-Listing zu PROJ-4-2: util.h

```
#ifndef _util_h
#define _util_h


struct buffer {
        unsigned char *buf;
        unsigned int bufmax;
        unsigned int buflen;
};




/* Buffer Functions
 * createBuf creates a new buffer (including malloc)
 * copyBuf copies the contents from src to dest (overwriting dest)
 * flushBuf empties a buffer (sets buflen to 0)
 * freeBuf frees the malloced memory
 *
 * returns -1 on error, 1 on success (for non-void functions)
*/
int createBuf(struct buffer *buf, unsigned int maxsize);
int copyBuf (struct buffer *dest, struct buffer *src);
void flushBuf(struct buffer *buf);
void freeBuf(struct buffer *buf);


#endif
```

Hinweise

- `createBuf()`
  ... initialisiert einen Puffer mit der Größe `maxsize`. Dazu muss ein Speicherbereich dieser Größe mittels der Funktion `malloc()` reserviert werden und die beiden Werte `bufmax` und `buflen` gesetzt werden.
  Rückgabewert: -1 im Fehlerfall, 1 sonst
- `copyBuf()` ... kopiert einen Puffer `src` nach `dest`. Der Puffer `dest` wird dabei, wenn möglich, überschrieben. Verwenden Sie die Funktion `memcpy()` und überlegen Sie sich, ob und welche Probleme und Fehler auftreten können.
  Rückgabewert: -1 im Fehlerfall, 1 sonst
- `flushBuf()` ... verwirft den momentanen Inhalt des Buffers.
  Rückgabewert: keiner
- `freeBuf()` ... soll einen Puffer mittels der Funktion `free()` wieder freigeben.
  Rückgabewert: keiner

Code-Listing zu PROJ-4-6: tokenizer.c.rahmen

```
int getTokenFromStream(int fd, struct buffer *buf, struct buffer *token, ...) {
        va_list ap, ap_copy;
        int r, t, retval = 1;
        int ts, tl, fl;

        struct pollfd pollfds[1];
        int pollret;


        pollfds[0].fd = fd;
        pollfds[0].events = POLLIN;



        va_start(ap, token);


        while(1) {
                va_copy(ap_copy, ap);
                t = searchToken(&ts, &tl, &fl, buf, ap_copy);
                va_end(ap_copy);


                if(t == 1) break;

                /* there won't be a token in this buffer anymore as it's full */
                if(buf->buflen >= buf->bufmax) {
                        retval = -1;
                        break;
                }

                if((pollret = poll(pollfds, 1, -1)) < 0) {
                        if(errno == EINTR) continue;
                        else {
                                retval = -1;
                                break;
                        }
                } else if (pollfds[0].revents & POLLIN) {
                        if((r = readToBuf(fd, buf)) == -1) {
                                retval = -1;
                                break;
                        } else if (r == 0) {
                                retval = 0;
                                break;
                        }
                } else {
                        /* unknown error */
                        retval = -1;
                        break;
                }
        }

        va_end(ap);

        if(t == 1) if(extractToken(buf, token, ts, tl, fl) == -1) return -1;

        return retval;
}
```

```
int getTokenFromBuffer(struct buffer *buf, struct buffer *token, ...) {
        va_list ap;
        int ts, tl, fl;

        /* we just don't care if searchToken thinks it found something or not */
        va_start(ap, token);
        searchToken(&ts, &tl, &fl, buf, ap);
        va_end(ap);


        if(tl == 0) return 0;
        else {
                if(extractToken(buf, token, ts, tl, fl) == -1) return -1;
        }

        return 1;
}
```

```
int getTokenFromStreamBuffer(struct buffer *buf, struct buffer *token, ...) {
        va_list ap;
        int t;

        int ts, tl, fl;

        va_start(ap, token);
        t = searchToken(&ts, &tl, &fl, buf, ap);
        va_end(ap);


        if(t == 1) if(extractToken(buf, token, ts, tl, fl) == -1) return -1;

        return t;
}
```

Hinweise

- Gegeben sind die drei einfachen Funktionen, die die Anwendung des Tokenizers (`searchString()` und `extractToken()`) erleichtern. Jede dieser Funktionen deckt einen der folgenden Aufgabenbereiche ab:
  - `int getTokenFromStream(int fd, struct buffer *buf, struct buffer *token, ...);`
    Lesen und zerteilen von Daten aus einem Datenstrom (Filedeskriptor): Der Datenstrom ist natürlich nicht abgeschlossen, so dass um ein Token zu finden auf jeden Fall ein Seperator auftreten muss. Aus dem Filedeskriptor wird nur gelesen, wenn kein Token gefunden wird und die Funktion kehrt erst zurück, wenn entweder ein Token gefunden wird, der Stream am Ende ist, oder ein Fehler auftritt. Deshalb kann diese Funktion nicht verwendet werden, wenn gleichzeitig auf verschiedenen Filedeskriptoren Ereignisse auftreten können (siehe auch `poll()`-Mechanismus auf dem nächsten Blatt).
  - Lesen und zerteilen von Daten aus einem Puffer (`struct buffer`)
    Hierbei gilt es zwei Fälle zu unterscheiden:

* `int getTokenFromStreamBuffer(struct buffer *buf, struct buffer *token, ...);`
  Wird, wie im vorherigen Fall angedeutet, auf mehrere Ereignisse gleichzeitig gewartet, kann man bei dem Eintritt eines Ereignisses genau 1x von einem Filedeskriptor lesen. Dabei gilt jedoch zu beachten, dass dies nicht unbedingt bedeutet, dass auch ein ganzes Token ankommt; umgekehrt können aber auch mehrere Token gleichzeitig ankommen. Dies macht die Trennung von Lesen aus dem Filedeskriptor und Zerteilen der Daten notwendig. Es wird also eine Funktion benötigt, die Daten zerteilt, die zuvor in einen Puffer gelesen wurden. Dieser Puffer hat zwar nur einen gewissen Füllstand, muss aber trotzdem, wie ein Datenstrom, als nicht abgeschlossen angesehen werden. Deswegen soll auch diese Funktion nur dann ein Token finden, wenn ein Separator gefunden wird.
* `int getTokenFromBuffer(struct buffer *buf, struct buffer *token, ...);`
  Da ein schon gefundenes Token noch weiter zerlegt werden können soll (also z.B. eine Zeile in einzelne Wörter), wird noch eine Funktion, die auf Puffern arbeitet, die nicht direkt aus einem Datenstrom kommen, sondern schon einmal durch den Tokenizer gelaufen und damit abgeschlossen sind, benötigt. Das Ende des Puffers zählt also sozusagen mit zu den Separatoren, dies muss aber gesondert behandelt werden.

  Diese beiden Funktionen lesen keine neuen Daten aus und können somit auch zurückkehren ohne ein Token zu finden (Rückgabewert 0) und blockieren die Programmausführung nicht.

- Die Rückgabewerte dieser Funktionen sind grundsätzlich dieselben wie beim Tokenizer, also -1 für Fehler, 0 für kein Token gefunden, 1 für ein gefundenes Token. Bei der Funktion `int getTokenFromStream()` bedeutet 0, dass der Filedeskriptor geschlossen wurde (Stream-Ende).

Hinweis zu PROJ-4-7:
Die Aufteilung in `basedir` und `subdir` ist notwendig, weil immer der relative Pfad zum `basedir` ausgegeben werden soll und nicht der absolute Pfad von der Wurzel des Dateisystems aus. Die Funktion `parseDirToFD()` wird normalerweise nur mit leerem String `subdir` aufgerufen werden - dieser ist nur intern für das Absteigen in Unterverzeichnisse nötig.

Hinweis zu PROJ-4-8:
In C gibt es eine spezielle Pointer-Arithmetik. Rechnet man mit Pointern (um neue Adressen zu erzeugen), so entspricht eine Zahl nicht Bytes, sondern Einheiten des Pointertyps. Das heißt: Ist `pointer` ein Zeiger vom Typ `int *` (also 32 Bit, bzw 4 Byte), der auf die Adresse `0x00000010` zeigt, so erzeugt `pointer + 1` nicht den Wert `0x00000011` sondern `0x00000014`. Damit zeigt der neue Wert auf die nächste Speicherstelle. Da wir hier aber mit nicht typisierten `void *`-Zeigern arbeiten müssen, da wir verschiedene Datenstrukturen in das Array packen wollen, gilt hier wieder die Byte-Arithmetik wie man es von normalen Zahlen erwarten würde.