

CONCURRENT PROGRAMMING IN C

TCP-Fileserver

Seminararbeit FS 2014

Student: Micha Schönenberger

Dozent: Nico Schottelius

© 2014

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Verzeichnis der Listings	V
1. Versionierung	1
2. Aufwände	2
3. Einleitung	5
3.1. Rahmenbedingungen	5
3.1.1. Geplante Termine	5
3.1.2. Administratives	5
3.1.3. Abgabebedingungen	5
3.1.4. Vortrag / Präsentation	6
3.1.5. Lernziele	6
3.1.6. Lerninhalte	7
3.2. Das Projekt	7
3.3. Ausgangslage	8
4. Anleitung zur Nutzung des Servers und des Clients	9
4.1. Starten des Server	9
4.2. Starten des Clients	11
5. Umsetzung des Projektes	12
5.1. Voraussetzungen	12
5.2. Libraries	12
5.3. Programmierumgebung	13
5.4. LOG/DEBUG	13
5.5. Speicherverwaltung – Buddy System	14
5.5.1. Wie wird das komplette System gemanagt?	14
5.5.2. Wie wird der optimale Block für ein neues File im Shared Memory gefunden?	15
5.5.3. Gibt es nur zu grosse Blöcke, wie werden die geteilt?	16
5.6. Locks	17

5.7. Files und deren Funktionen	18
6. Fazit	19
A. Anhang	i
A.1. Screenshots zum Server	i
A.1.1. Shared Memory vor dem Einfügen eines neuen Files	i
A.1.2. Aufteilen des Shared Memory in der Funktion devide()	ii
A.1.3. Shared Memory nach dem Einfügen eines neuen Files	iii
A.2. Programm-Code	iv
A.2.1. Server.c	iv
A.2.2. Client.c	xvii
Literaturverzeichnis	xxii

Abbildungsverzeichnis

4.1. Starten des Servers	10
4.2. Starten des Servers mit doppelten Argumenten	10
4.3. Starten des Clients	11
A.1. SHM vor dem Einfügen eines neuen Files	i
A.2. Aufteilen des SHM in der Funktion devide()	ii
A.3. SHM nach dem Einfügen eines neuen Files	iii

Tabellenverzeichnis

1.1. Versionierung Dokumentation	1
2.1. Aufwände Seminararbeit	4
3.1. geplante Termine	5
5.1. Loglevels	13
5.2. Shared Memory control Struct	15
5.3. Shared Memory - 1	16
5.4. Shared Memory - 2	16
5.5. bar	18
5.6. myfunctions.c	18

Verzeichnis der Listings

5.1. itskylib.c	12
5.2. shm_ctr_struct	14
A.1. Server.c	iv
A.2. Client.c	xvii

1. Versionierung

Version	Datum	Beschreibung
V0.1	15.03.2014	Ersterstellung Dokument
V0.2	17.03.2014	Einleitung, Ausgangslage
V0.3	07.04.2014	Grundgerüst, Konzept
V0.4	08.04..2014	Implementierung Argument-Überprüfung (LogLevel)
V0.5	13.04.2014	Erweitern Server (Argument-Überprüfung)
V0.6	01.05.2014	Speicherverwaltung mit Buddy
V0.7	02.05.2014	Client Connection to Server
V0.8	02.05.2014	TCP Connection Protocol, Loglevel
V0.9	02.05.2014	CREATE, LIST, LOG, Fehlebehebungen
V1.0	02.05.2014	One Thread per Client, Refactoring, Commenting Code
V1.1	02.05.2014	Client Connection to Server
V1.2	02.05.2014	Implementing RWLock for Creating File
V1.3	03.05.2014	
V1.3b	04.05.2014	
V1.3c	05.05.2014	
V1.4	06.05.2014	
V1.5	26.05.2014	
V1.5b	27.05.2014	
V1.5c	28.05.2014	
V1.6	31.05.2014	
V1.6b	01.06.2014	
V1.7	02.06.2014	
V1.8	10.06.2014	

Tabelle 1.1.: Versionierung Dokumentation

2. Aufwände

Datum	Zeit	Beschreibung
15.03.2014	3.75h	<ul style="list-style-type: none"> -Ersterstellung Dokumentation -Github Repo erstellen -Einlesen Buch Kapitel 15 (Semaphore, Shared Memory, ...) -Erstellen Debian VM
17.03.2014	0.5h	-Dokumentation: Einleitung (Rahmenbedingungen, Projekt, Ausgangslage)
07.04.2014	1.75h	-Grundgerüst erstellen, LOG-LEVEL definieren
08.04.2014	3h	<ul style="list-style-type: none"> -Parsing Argumente bei Programmstart -Log-Level Implementierung
13.04.2014	1.75h	<ul style="list-style-type: none"> -Auslagern Funktionen in externe .h Dateien -Anpassen Argument-Validierung: wenn Argument mehr als 1mal vorkommt, wird es ignoriert -bei nicht setzen des LogLevel wird default LogLevel initialisiert -Erstinitialisierung TCP-Server: wartet auf Verbindung von Client -Probleme: #define von LOG LEVELS in log-Level.h sind nicht sichtbar in "server.h".
15.04.2014	1h	-Installieren von e-UML -> funktioniert nur mit Java ;-(
01.05.2014	9h	<ul style="list-style-type: none"> -Debug mit #define funktioniert nicht. -> Einlesen in andere Möglichkeiten -gemäss Rücksprache mit anderen Studenten sollte nicht ein File wirklich eingelesen werden (von HDD geöffnet und Stream übermittelt), sondern lediglich mit dem Filenamen und Grösse angelegt werden im Shared Memory - Versuch, Control Shared Memory zu lösen mit einem Buddy System Fazit Arbeiten: - Server startet ohne Fehler -Loglevel gelöscht (da nicht funktionstüchtig) -Port kann mit Argument "-p" mitgegeben werden -bei starten des Servers ohne Argumente kommt die Hilfeseite -Das Kontroll-Strukt für das Shared Memory ist implementiert.

		-Die Speicherverwaltung mit Buddy-System wurde beschlossen. Das aufteilen der Blöcke funktioniert einwandfrei (wieder vereinen ist noch nicht implementiert)
02.05.2014	2.25h	-Client TCP Connection zu Server aufbauen -Client kann Verbindung aufbauen, Message senden und Message erhalten. Es fehlt jedoch ein Protokoll, dass die Übertragung sicherstellt. -Teils werden noch zusätzlich Zeichen angezeigt (z.B. 25\$?d anstelle von 25) -es gibt noch keine Validierung der Argumente (z.B. CREATE, DELETE, ...)
03.05.2014	6h	-Log-Level implementiert mit verschiedenen Stufen. Output momentan nur möglich auf CLI, jedoch mit Datum (z.B. May 3 2014 15:37:15: WARNING Test Log Warning) - Implementierung von kleinem TCP Protokoll (funktioniert nur beim Senden Client)
04.05.2014	6h	- Überprüfung 1. Wort von Client als Command-Argument (Momentan nur Create File) -Verfeinern CREATE Command - LOG verbessern
05.05.2014	8h	-beim CREATE vom 2. File wurde der Name des ersten Files überschrieben. Stundenlange Suche nach Ursache (Problem war ein zuweisen eines Pointer zum andren filename = filename.new anstelle filename = strdup(filename.new) - Implementierung von LIST shm, was dem Client eine komplette Liste des Shared Memory mit Adresse und Dateinamen zurückliefert.
06.05.2014	5.5h	-Dokumentation letzte 2 Tage - Beheben von Warnings beim Kompilieren -für das TRACE_LOG können nur mehrere (dynamische Variablen mitgeliefert werden. -Problem, dass Server teils beim Erstellen eines Files abstürzt. Recherche im Internet: 1 Fehler war das malloc vor einen strdup() -> Weniger Abstürze, aber nicht ganz weg - DELETE und READ fertig implementieren (ohne Lock)
26.05.2014	1.75h	-Erstellen von Pthreads für Clients
27.05.2014	4.25h	-Erstellen von PThreads für Client - Implementierung ReadWrite Lock mit pthread_rwlock_t (Momentan nur ReadLock beim Lesen) -Kommentieren von Code

		<ul style="list-style-type: none"> -löschen von altem, nicht mehr benutztem Code -Anpassen Log-Design (damit besser lesbar)
28.05.2014	2h	<ul style="list-style-type: none"> -Update Dokumentation -Anpassen Version Github/Dokumentation -Kapitel 5.7 beginnen
31.05.2014	4.5h	<ul style="list-style-type: none"> -Fehlerbehebung PThreadList - Senden von EXIT bei Beenden von Client an Server - Joining PThread nach Client-EXIT bei Server
01.06.2014	7h	<ul style="list-style-type: none"> - Joining PThread nach Client-EXIT bei Server fertig -Probleme Segmentation Fault beim löschen des letzten Files (mehrere Stunden Fehlersuche) -> Problem war Test.txt (fix in Code als Testfile) -Implementierung von RW-Lock bei DELETE File
02.06.2014	3.75h	<ul style="list-style-type: none"> -Fehlerbehebung bei Übermittlung von grösseren Fileinhalten -Code kommentieren -Dokumentation erweitern
10.06.2014	99h	<ul style="list-style-type: none"> -Dokumentation anpassen und erweitern -Server Parameter optimieren (-p, -l) -Kapitel 4

Tabelle 2.1.: Aufwände Seminararbeit

3. Einleitung

3.1. Rahmenbedingungen

Die Aufgabenstellung und die Rahmenbedingungen wurden über Github (https://github.com/telmich/zhaw_seminar_concurrent_c_programming) veröffentlicht.

Anbei ein Auszug aus den wichtigsten Eckdaten und Anforderungen:

3.1.1. Geplante Termine

Datum	Beschreibung
13.03.2014	Kick-Off Meeting
16.03.2016	Abgabe der schriftlichen Arbeit (1 Woche vor Präsentation)
22.06.2014	Präsentation der Arbeit
01.07.2014	Präsentation der Arbeit
02.07.2014	optionale Teilnahme an anderen Präsentationen
03.07.2014	optionale Teilnahme an anderen Präsentationen
21.07.2014	Notenabgabe

Tabelle 3.1.: geplante Termine

3.1.2. Administratives

- Abgabe Arbeit via git repository auf github.com
- Zum Zeitpunkt "Abgabe Arbeit" werden alle git repositories geklont, Änderungen danach werden *NICHT* für die Benotung beachtet.

3.1.3. Abgabebedingungen

- git repo auf github vorhanden
- Applikation lauffähig unter Linux
- Nach "make" Eingabe existiert
 - "run": Binary des Servers
 - Sollte nicht abstürzen / SEGV auftreten

- “test“: Executable zum Testen des Servers
- “doc.pdf“: Dokumentation
- Einleitung
- Anleitung zur Nutzung
- Weg, Probleme, Lösungen
- Fazit
- Keine Prosa - sondern guter technischer Bericht
- Deutsch oder English möglich

3.1.4. Vortrag / Präsentation

- 10 - 15 Minuten + 5 Minuten Fragen
- Richtzeiten:
- Einleitung (2-3) min
- Weg, Probleme, Lösungen (4-10) min
- Implementation zeigen (2-5) min
- Fragen (2-5) min
- Vortrag ist nicht (nur) für den Dozenten

3.1.5. Lernziele

- Die Besucher des Seminars verstehen was Concurrency bedeutet und welche Probleme und Lösungsansätze es gibt.
- Sie sind in der Lage Programme in der Programmiersprache C zu schreiben, die auf gemeinsame Ressourcen gleichzeitig zugreifen.
- Das Seminar setzt Kenntnisse der Programmiersprache C voraus.

3.1.6. Lerninhalte

- Selbstständige Definition des Funktionsumfangs des Programmes unter Berücksichtigung der verfügbaren Ressourcen im Seminar.
- Konzeption und Entwicklung eines Programms, das gleichzeitig auf einen Speicherbereich zugreift.
- Die Implementation erfolgt mithilfe von Threads oder Forks und Shared Memory (SHM).

3.2. Das Projekt

- kein globaler Lock (!)
- Kommunikation via TCP/IP (empfohlen) - Wahlweise auch Unix Domain Socket
- fork + shm (empfohlen)
 - oder pthreads
 - für jede Verbindung einen prozess/thread
 - Hauptthread/prozess kann bind/listen/accept machen
- Fokus liegt auf dem Serverteil
 - Client ist hauptsächlich zum Testen da
 - Server wird durch Skript vom Dozent getestet
- Wenn die Eingabe valid ist, bekommt der Client ein OK
 - Locking, gleichzeitiger Zugriff im Server lösen
 - Client muss *nie* retry machen
- Protokolldefinitionen in protokoll/
- Alle Indeces beginnen bei 0
- Debug-Ausgaben von Client/Server auf stderr

Fileserver

- Dateien sind nur im Speicher vorhanden
- Das echte Dateisystem darf NICHT benutzt werden
- Mehrere gleichzeitige Clients
- Lock auf Dateiebene

3.3. Ausgangslage

Die Aufgabenstellung, wie sie oben beschrieben ist, ist für einen nicht Programmierer gemäss Dozent eine grosse Herausforderung. Mindestens vier Studenten, zu denen auch ich zähle, haben ihre Bedenken geäussert, dass diese Aufgabenstellung fast nicht zu erreichen ist. Ein Informatiker, dessen Zuhause ist das Programmieren ist geschweige denn die Sprache "C", wird für eine minimalistische Lösung bei weitem mehr Stunden benötigen als die 60 Stunden, welche für dieses Seminararbeit gedacht sind.

Damit für den Dozenten besser ersichtlich ist, wie viel Zeit aufgewendet wurde und für welche Teile der Arbeit, werden im Kapitel 2 (Aufwände) die Zeiten erfasst und ausgewiesen.

4. Anleitung zur Nutzung des Servers und des Clients

Dieses Kapitel widmet sich mit dem Umgangs des Servers und der dazugehörigen Clients. Angefügte Screenshots sollen einen Einblick geben, wie die Software funktioniert, auch wenn kein Computer zum Austesten vorhanden ist.

4.1. Starten des Server

Der Server ist das Herzstück der Applikation. Er ist so ausgelegt, dass er mit gültigen Argumenten erweitert werden kann.

Momentan gibt es zwei implementierte Argumente, welche beim Start mitgegeben werden können:

- «-l» – Loglevel
Für das Loglevel gültige Eingaben sind Integer mit Werten von 0-7.
Wird ein ungültiger Wert grösser als 8 eingegeben, wird der Fehler abgefangen und das Loglevel wird auf den default-Wert = 5 gesetzt.
- «-p» – Serverport Für den Server-Port gültige Eingaben sind: 1024 - 65535. Die well-known Ports von 1 - 1023 wurden bewusst nicht erlaubt, da es Konflikte geben könnte mit anderen Applikationen.
Wird ein ungültiger Wert eingegeben, wird der Server automatisch mit dem default-Port = 7000 starten.

Beispiele für gültiges Starten des Servers:

```
./Server -p 4637 -l 7  
./Server -l 6 -p 5479  
./Server -p 7788  
./Server -l 8
```

Die Abbildung 4.1 zeigt einen gültigen Serverstart. Bei der Abbildung 4.2 ist zu sehen, wie das Loglevel 8 nicht gesetzt werden kann und wie die nachträglichen Argumente (Duplikate) des Serverports und des Loglevels ignoriert werden.

Anmerkung:

Die Reihenfolge der Argumente ist egal. Wird ein Argument zwei mal eingegeben (z.B.

```

parallels@ubuntu:Program> ./Server -p 8877 -l 8
Setting up shared Memory ...Verify valid arguments ...

-----
Argument No. 1 Value = -p      -> Hit for Server-Port

Set up now Server Port to 8877 ...      ... Done

-----
Argument No. 3 Value = -l      -> Hit for Loglevel

LogLevel not valid [1-8]. Will set Log-Level to 5 ... Done
Set up TCP-Server settings ...
# Jun 10 2014 15:13:35: LOG_ALERT          Server is now going to Listening Mode for Clients.
# Jun 10 2014 15:13:35: LOG_ALERT          Client can connect to Server on Port 8877

```

Abbildung 4.1.: Starten des Servers
Quelle: eigener Screenshot

```

parallels@ubuntu:Program> ./Server -p 8877 -l 8 -p 4433 -l 4
Setting up shared Memory ...Verify valid arguments ...

-----
Argument No. 1 Value = -p      -> Hit for Server-Port

Set up now Server Port to 8877 ...      ... Done

-----
Argument No. 3 Value = -l      -> Hit for Loglevel

LogLevel not valid [1-8]. Will set Log-Level to 5 ... Done

-----
Argument No. 5 Value = -p      -> Hit for Server-Port

ServerPort was already set. New argument will be ignored.

-----
Argument No. 7 Value = -l      -> Hit for Loglevel

LogLevel was already set. New argument will be ignored.
Set up TCP-Server settings ...
# Jun 10 2014 15:13:35: LOG_ALERT          Server is now going to Listening Mode for Clients.
# Jun 10 2014 15:13:35: LOG_ALERT          Client can connect to Server on Port 8877

```

Abbildung 4.2.: Starten des Servers mit doppelten Argumenten
Quelle: eigener Screenshot

./Server -p 7524 -l 6 -l 8), wird nur das erste Argument berücksichtigt. Alle weiteren Argumente werden ignoriert.

./Server -p 5432 -l 5 -l 8 wird somit den TCP Port 5432 und das Loglevel 5 setzen.

./Server -p 2066 -p 5432 -l 4 -l 8 wird somit den TCP Port 2066 und das Loglevel 4 setzen.

4.2. Starten des Clients

Der Client selber besitzt im Gegensatz zum Server eine sehr eingeschränkte Logik. Seine Aufgabe besteht hauptsächlich darin, sich über einen TCP-Socket mit dem Server zu verbinden.

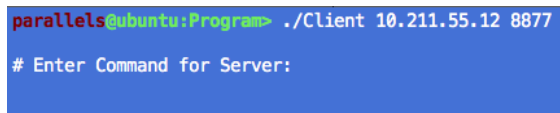
Beim Client ist es notwendig, die Argumente in der richtigen Reihenfolge einzugeben. So ist nur folgender Aufruf gültig:

```
./Client <IP Adresse von Server>
```

oder

```
./Client <IP Adresse von Server> <TCP Port Server>
```

Wird die Variante ohne den Port gewählt, versucht sich der Client über den default Port = 7000 mit dem Server zu verbinden.



```
parallels@ubuntu:Program> ./Client 10.211.55.12 8877  
# Enter Command for Server:
```

Abbildung 4.3.: Starten des Clients
Quelle: eigener Screenshot

Beispiele für gültiges Starten des Clients:

```
./Client 10.211.55.12
```

```
./Client 10.211.55.12 5542
```

5. Umsetzung des Projektes

5.1. Voraussetzungen

Da der Student kein Programmierer ist und nur schulische Kenntnisse von der Programmiersprache Java besitzt, wird dieses Projekt eine grosse Herausforderung. Deshalb soll das Grundkonzept als Stütze dienen, so dass sich der Programmierer nicht in den Details verlieren soll.

5.2. Libraries

Im Unterricht des Modules «Systemsoftware» wurden verschiedene Libraries durch den Dozenten zur Verfügung gestellt.

Diese sollen, da sie einige Grundfunktionen wie das Error-Handling bereits beinhalten, in diesem Projekt ebenfalls genutzt werden. Die so genutzten Dateien werden nicht explizit als Quelle erwähnt. Sie besitzen jedoch im Kopf die Daten des Dozenten und sind als externe Datei erkennbar. Als Beispiel zeigt das Listing 5.1 die Anbindung einer externen Datei.

```
1 /* (C) IT Sky Consulting GmbH 2014\  
2  * http://www.it-sky-consulting.com/\  
3  * Author: Karl Brodowsky\  
4  * Date: 2014-02-27\  
5  * License: GPL v2 (See https://de.wikipedia.org/wiki/GNU\_General\_Public\_License  
6    )\  
7  *\  
8  * This file is inspired by\  
9  * http://cs.baylor.edu/~donahoo/practical/CSockets/code/HandleTCPClient.c\  
10 */
```

Listing 5.1: itskylib.c

5.3. Programmierumgebung

Programmiert wird auf einem MAC OS-X 10.9 (Mavericks). Die eingesetzte Software ist das Eclipse mit dem integrierten «Eclipse C/C++ Development Tools». Eclipse ist bereits aus der Java-Programmierung im Grundstudium bekannt und eingerichtet. So musste nur noch die «Eclipse C/C++ Development Tools» installiert werden. Der grosse Vorteil gegenüber eines Texteditors ist das Auto-Complete und die automatische Formatierung des Codes. Für das Kompilieren und Ausführen des Codes wird eine Ubuntu genutzt. Dieses ist als virtuelle Maschine über Parallels installiert. Zugriffen auf das Ubuntu wird mittels SSH von MAC OS-X. Der Grund, Ubuntu zu nutzen liegt in den anderen Bibliotheken, welche teils in MAC OS-X nicht genutzt werden können oder anders implementiert sind. Ebenfalls aufgefallen im Unterricht war, dass Ubuntu 32-bit und Ubuntu 64-bit nicht immer gleich implementiert sind.

Eckdaten Ubuntu:

- OS: ubuntu 12.04 LTS
- Memory: 900 MB
- CPU: Intel Core i7-2677M CPU @ 1.80 GHz
- OS-Type: 64bit

5.4. LOG/DEBUG

Die Implementierung des LOG soll als Erstes geschehen. So soll sichergestellt werden, dass während der Programmierung das LOG-Level geändert werden kann und allfällige Fehler schneller gesehen werden können.

Die Definition der LOG-Levels wird analog zu den syslog LOG-Level erstellt:

LEVEL	Bezeichnung
0	EMERGENCY
1	ALERT
2	CRITICAL
3	ERROR
4	WARNING
5	NOTICE
6	INFORMATIONAL
7	DEBUG

Tabelle 5.1.: Loglevels

5.5. Speicherverwaltung – Buddy System

Für die Verwaltung des Shared Memory (shm) bedarf es einer Logik, um die verschiedenen Adressen im Shared Memory richtig ansprechen zu können. Zusätzlich muss sichergestellt werden, dass kein File in das shm geschrieben wird, dass länger ist als der freie Speicherplatz, bevor das nächste File kommt.

Es gibt viele Dokumentierte Speicherverwaltungen. Nach längerer Recherche wurde entschieden, dass der Speicher mit dem Buddy-System verwaltet werden soll. Die Suche im Internet nach einer vorhandenen Library für die Speicherverwaltung mit dem Buddy-System blieb leider erfolglos. Also blieb nichts anderes übrig, als das Buddy-System von grund auf selber zu gestalten und zu implementieren.

Dabei wurden sehr viele Fragen aufgeworfen, welche Schrittweise erarbeitet wurden

5.5.1. Wie wird das komplette System gemanagt?

Für das Management des shared Memory wurde ein Struct erstellt (siehe Listing 5.2, welches das Shared Memory kontrollieren soll.

```
1 struct shm_ctr_struct {  
2     int shm_size; //size of shm-block  
3     int isfree ; // indicates if block is free or not  
4     int isLast; //indicates the end of shared memory  
5     struct shm_ctr_struct *next;  
6     struct shm_ctr_struct *prev;  
7     char *filename;  
8     char *filedata ; // just this pointer is a pointer to Shared memory  
9     pthread_rwlock_t rwlockFile; //Read-write lock for file  
10 };
```

Listing 5.2: shm_ctr_struct

Folgende Tabelle soll aufzeigen, welches Attribut im Struct welche Funktion übernehmen soll. Das Ziel des Structs ist eine verkettete Liste. Das erste Struct ist im Main global bekannt, das letzte wird gefunden, da isLast auf TRUE gesetzt ist.

Struct Attribut	Bezeichnung
int shm_size	Grösse des Blockes des Shared Memory Bereiches
int isLast	TRUE wenn es der letzte Block ist, sonst FALSE
int isfree	TRUE wenn Block frei ist, FALSE wenn Block besetzt ist
struct shm_ctr_struct *next	Pointer auf den nächsten Block (zeigt auf sich selber, wenn es der letzte Block ist)
struct shm_ctr_struct *prev	Pointer auf den vorherigen Block (implementiert, aber nicht benutzt)
char *filename	Pointer auf den Dateinamen, der im Block gespeichert ist (NULL wenn kein File gespeichert ist)
char *filedata	Dies ist der einzige Pointer auf das Shared-Memory. Hier liegen die effektiven Daten des Files.
pthread_rwlock_t rwlockFile	Für jede Instanz des Structs und somit für jedes unique File wird ein ReadWrite-Lock erstellt.

Tabelle 5.2.: Shared Memory control Struct

5.5.2. Wie wird der optimale Block für ein neues File im Shared Memory gefunden?

Hierzu wurde die Funktion `find_shm_place(...)` erstellt.

Diese Funktion beginnt beim ersten Eintrag des Structs `shm_ctr_struct` (siehe Listing 5.2) und sucht über alle vorhanden Blöcke (über den next-Pointer) einen optimalen Block. Optimal bedeutet, dass er grösser oder gleich der Grösse der neu zu erstellenden Dokumentes sein muss, aber nicht grösser als das doppelte. Wäre er grösser als das Doppelte, wäre das Speicherplatzverschwendung. Zusätzlich muss er frei sein (`isfree = TRUE`).

5.5.3. Gibt es nur zu grosse Blöcke, wie werden die geteilt?

Das Buddy-System gibt vor, dass die Blockgrößen aus 2er Potenzen gebildet werden. Also 2, 4, 8, 16, 32 ...

Beispiel Buddy-System:

Shared Memory – SIZE = 65535

Tabelle 5.3.: Shared Memory - 1

Ist die Dateigröße = 14547, gibt es keinen optimalen Block. Der optimale Block wäre hier 2^{14} (= 16384). Der kleiner Block 2^{13} (= 8192) wäre hier zu klein. Zuerst muss müssen nun die Blöcke aufgeteilt werden, so dass folgende Blöcke entstehen:

Block 1 SIZE = 16384	Block 2 SIZE = 16384	Block 3 SIZE = 32768
-------------------------	-------------------------	-------------------------

Tabelle 5.4.: Shared Memory - 2

Für die Aufteilung wurde die Funktion `devide(...)` implementiert.

Diese beginnt beim ersten Block und arbeitet sich (über den next-Pointer) nach hinten. Beim ersten gefundenen freien Block, wird nun die Block-Size halbiert. Es wird ein neuer Block erzeugt und die Verlinkungen (next, previous, Pointer auf Filename und Filedata sowie isFree und size) werden dem bestehenden und neuen Block gesetzt, so dass die Linked-List wieder komplett vorhanden ist.

Ist die Blockgröße die gewünschte Größe, findet ein `return = TRUE` statt. Ansonsten wird die Funktion selber rekursiv aufgerufen, bis die Blockgröße genügend klein ist. Dann erfolgt der `return = TRUE`. Ein Screenshot der Funktion `devide()` ist im Anhang [A.2](#) zu finden. Ebenfalls im Anhang [A.1](#) und [A.3](#) ist das Shared Memory vor und nach dem Einfügen eines neuen Files zu sehen.

5.6. Locks

Wie im Kapitel 3.2 erwähnt, ist ein global Lock nicht erlaubt.

Für die Umsetzung des Locks wurde schlussendlich kein mutex gewählt, wie anfangs gedacht war. Das Problem beim mutex ist, dass ein lesender Client das ganze File ebenfalls sperrt für weitere Lesezugriffe. Dies soll jedoch nicht der Fall sein.

Aus diesem Grund wurde auf «pthread_rwlock_t» zurückgegriffen.

Die Implementation des Locks wurde gemäss Tabelle im Kapitel 5.5.1 vorgenommen.

Da das Kontroll-Strukt für das Shared-Memory bereits vorhanden war, konnte «pthread_rwlock_t» ohne Probleme eingefügt werden.

Soll nun ein File gelockt werden, kann das elegant gelöst werden:

- Wenn das File gelesen werden möchte, muss zwingend die Adresse des entsprechenden Strukt bereits vorhanden sein = struct shm_ctr_struct *shm_ctr
- Nun kann ein ReadLock über pthread_rwlock_rdlock(&(shm_ctr->rwlockFile)); gemacht werden.

5.7. Files und deren Funktionen

Um die auf den ersten Blick nicht ganz klare Strukturen aufzeigen zu können, soll sich dieses Kapitel mit den einzelnen Files beschäftigen, die für den Server und den Client notwendig sind. Jede Funktion jedes Files soll kurz und bündig erläutert werden.

myfunctions.c
beinhaltet eigene definierte Funktionen
getFixCharLen(char *mychar, int mylength) füllt einen CharPointer bis zur gewählten Länge auf. Wird benötigt für schöne Darstellung im LOG void print_all_shm_blocks(struct shm_ctr_struct *shm_ctr) Gibt auf der Konsole alle Blöcke des SHM aus. Wird zu DEBUG-Zwecken benötigt char * get_all_shm_blocks(struct shm_ctr_struct *shm_ctr) Gibt alle Blöcke des SHM als char Pointer zurück. Wird benötigt, um Client das SHM zu übermitteln void print_single_shm_blocks(struct shm_ctr_struct *shm_ctr) Gibt auf der Konsole einen Block des SHM aus. Wird zu DEBUG-Zwecken benötigt char * getSingleString(char *msg, ...) Gibt einen Char Pointer als Return Wert. Diese Funktion erlaubt es, einen ?String? mit Argumenten (z.B. %i, %s) zu übergeben. Diese werden zur Laufzeit interpretiert und als neuen Char Pointer zurückgegeben

Tabelle 5.5.: bar

Tabelle 5.6.: myfunctions.c

6. Fazit

FEHLT NOCH

A. Anhang

A.1. Screenshots zum Server

A.1.1. Shared Memory vor dem Einfügen eines neuen Files

```
# Message from Server: ===== ALL BLOCKS OF SHARED MEMORY =====
Block No 1:      Block-Address = 1b7c010      Block-Size = 2048      isFree = 0 Filename = Test.txt
Block No 2:      Block-Address = 1b7c4a0      Block-Size = 2048      isFree = 1 Filename = NULL
Block No 3:      Block-Address = 1b7c3a0      Block-Size = 4096      isFree = 1 Filename = NULL
Block No 4:      Block-Address = 1b7c2a0      Block-Size = 8192      isFree = 1 Filename = NULL
Block No 5:      Block-Address = 1b7c1a0      Block-Size = 16384     isFree = 1 Filename = NULL
Block No 6:      Block-Address = 1b7c0a0      Block-Size = 32768     isFree = 1 Filename = NULL
===== END OF SHARED MEMORY =====
```

Abbildung A.1.: SHM vor dem Einfügen eines neuen Files
Quelle: eigener Screenshot

A.1.2. Aufteilen des Shared Memory in der Funktion devide()

```

Want to write filename with size=1056 to shm
Adress of shm Place to check is 215f010
Size of shm Place is 65536
At the end of all shared memory places... No hit found to enter the filename.
Checked a good address is: 0
0 is not valid. So there is no good place to write the file into... Trying no to
Now in round_up_int(). Filesize needed = 1056 until = 16
i = 11 Output now set to = 2048 and return it.
Now in devide(). I want to devide until I reach Block size of 2048
=====
Address: d386b000      Block-Size = 32768      Filename = NULL
=====
Recursive call in deviding because block size is to big (at moment = 32768) ...
Now in devide(). I want to devide until I reach Block size of 2048
=====
Address: d386b000      Block-Size = 16384      Filename = NULL
=====
Recursive call in deviding because block size is to big (at moment = 16384) ...
Now in devide(). I want to devide until I reach Block size of 2048
=====
Address: d386b000      Block-Size = 8192      Filename = NULL
=====
Recursive call in deviding because block size is to big (at moment = 8192) ...
Now in devide(). I want to devide until I reach Block size of 2048
=====
Address: d386b000      Block-Size = 4096      Filename = NULL
=====
Recursive call in deviding because block size is to big (at moment = 4096) ...
Now in devide(). I want to devide until I reach Block size of 2048
After deviding I have a good block size.

Will now output all shm-blocks...
=====
Block No 1:      Block-Size = 2048      Filename = NULL      isLast = 0
Block No 2:      Block-Size = 2048      Filename = NULL      isLast = 0
Block No 3:      Block-Size = 4096      Filename = NULL      isLast = 0
Block No 4:      Block-Size = 8192      Filename = NULL      isLast = 0
Block No 5:      Block-Size = 16384     Filename = NULL      isLast = 0
Block No 6:      Block-Size = 32768     Filename = NULL      isLast = 1
=====

```

Abbildung A.2.: Aufteilen des SHM in der Funktion devide()
Quelle: eigener Screenshot

A.1.3. Shared Memory nach dem Einfügen eines neuen Files

# Message from Server: ===== ALL BLOCKS OF SHARED MEMORY =====			
Block No 1:	Block-Address = 1b7c010	Block-Size = 2048	isFree = 0 Filename = Test.txt
Block No 2:	Block-Address = 1b7c4a0	Block-Size = 2048	isFree = 1 Filename = NULL
Block No 3:	Block-Address = 1b7c3a0	Block-Size = 32	isFree = 0 Filename = test2.txt
Block No 4:	Block-Address = bc003210	Block-Size = 32	isFree = 1 Filename = NULL
Block No 5:	Block-Address = bc003110	Block-Size = 64	isFree = 1 Filename = NULL
Block No 6:	Block-Address = bc003010	Block-Size = 128	isFree = 1 Filename = NULL
Block No 7:	Block-Address = bc002f10	Block-Size = 256	isFree = 1 Filename = NULL
Block No 8:	Block-Address = bc002e10	Block-Size = 512	isFree = 1 Filename = NULL
Block No 9:	Block-Address = bc002d10	Block-Size = 1024	isFree = 1 Filename = NULL
Block No 10:	Block-Address = bc002c10	Block-Size = 2048	isFree = 1 Filename = NULL
Block No 11:	Block-Address = 1b7c2a0	Block-Size = 8192	isFree = 1 Filename = NULL
Block No 12:	Block-Address = 1b7c1a0	Block-Size = 16384	isFree = 1 Filename = NULL
Block No 13:	Block-Address = 1b7c0a0	Block-Size = 32768	isFree = 1 Filename = NULL

Abbildung A.3.: SHM nach dem Einfügen eines neuen Files

Quelle: eigener Screenshot

A.2. Programm-Code

A.2.1. Server.c

```

1  /*
2  * File:    Main.c
3  * Author:  Micha Schö̃nenberger
4  * Modul:   Concurrent Programming in C
5  *
6  * Created: 07.04.2014
7  * Project:  https://github.com/schoenm1/concurrent_c.git
8  */
9
10 /* ----- How to use this Program
11  -----
12  1) ...
13  2) ...
14  3) ...
15
16 -----
17  */
18 #define _XOPEN_SOURCE
19 #include "Logs.h"
20 #define TOT_SHM_SIZE 65536
21 #define MIM_SHM_BLOCK_SIZE 4
22 #define MAX_FILE_LENGTH 1500
23 #define MAX_WORD_SIZE 256
24 #define MUTEXSIZE 10
25 #include <arpa/inet.h> /* for sockaddr_in, inet_addr() and inet_ntoa() */
26 #include <errno.h>
27 #include <math.h>
28 #include <netinet/in.h>
29 #include <netinet/tcp.h>
30 #include <stdio.h> /* for printf() and fprintf() and ... */
31 #include <stdlib.h> /* for atoi() and exit() and ... */
32 #include <sys/socket.h> /* for socket(), bind(), recv, send(), and connect() */
33 #include <sys/types.h>
34 #include <unistd.h> /* for close() */
35 #include <stdarg.h>
36 #include <string.h>

```

```

36 #include <itskylib.h>
37 #include <time.h>
38 #include "mystructs.h" // global structs
39 #include "shm_control.c" // global structs
40 #include "pthread_control.c" // control of Pthreads
41 #include "myfunctions.c"
42 #include <pthread.h> /* for pthreads */
43 /*
44  =====
45  */
46 /* all needed for Shared Memory */
47 #include "shm.c"
48
49
50 /* all needed for handle Files */
51 #include "handleFiles.c"
52 #define BUFSIZE 8192 /* Size of receive buffer */
53 #define MAXPENDING 5 /* Maximum outstanding connection requests */
54 #define MAXRECWORDS 30000 /* Maximum of words receiving from the client */
55 #define SERVERPORT_ARG "-p";
56 #define LOGLEVEL_ARG "-l";
57 #define SERVERNAME "Server";
58 int LOGLEVEL = 4;
59 int _MAX_LENGTH_ARG = 5; // defines the maximum Length of arguments. e.g.
60     "_l"
61
62 struct pthread_struct *myPThreadStruct;
63
64 /* all global variables for Arguments */
65 char _logLevel_arg[10] = LOGLEVEL_ARG
66 ;
67 char _serverPort_arg[10] = SERVERPORT_ARG
68 ;
69
70 /* forward declarations of functions */
71 int setup_shm();
72 void my_handler(int signo);
73 int initshm(char *shm_start);

```

```

72 int setTCPServer();
73 void ServerListen();
74 void runClientCommand(char *recMessage[], char *command, int clntSocket, int
    thread_count);
75 void handle_tcp_client(void* parameters);
76 void breakCharArrayInWords(char *recMessage[], char *recBuffer[]);
77 int setLogLevel();
78
79 //int pthread_create(pthread_t * __restrict, const pthread_attr_t * __restrict, void
    *(*)(void *), void * __restrict);
80
81 /* global vars for TCP-Server */
82 int servSock; /* Socket descriptor for server */
83 int clientSocket; /* Socket descriptor for client */
84 struct sockaddr_in squareServAddr; /* Local address */
85 struct sockaddr_in ClientSocketAddress; /* Client address */
86 unsigned short ServerPort; /* Server port */
87 unsigned int client_address_len; /* Length of client address data structure */
88
89 struct shm_ctr_struct *shm_ctr;
90 struct validArgs {
91     int isSet;
92     char arg[10];
93 };
94
95 struct validArgs validArguments[5];
96
97 #include "valid-args.h"
98
99 int setup_shm() {
100     /* set up shared Memory */
101     printf("Setting up shared Memory ...");
102
103     /* create REF File, if it not exists */
104     remove(REF_FILE);
105     create_if_missing(REF_FILE, S_IRUSR | S_IWUSR);
106
107     /*create shm 'unique' key */
108     key_t shm_key = ftok(REF_FILE, 1);
109     if (shm_key < 0) {
110         handle_error(-1, "ftok failed", NO_EXIT);

```



```

111 }
112 create_shm(shm_key, "create", "shmget failed", IPC_CREAT | IPC_EXCL);
113 shm_id = create_shm(shm_key, "create", "shmget failed", 0);
114 return 1;
115 }
116
117 /* Handles the Signals which are received by the Client */
118 void my_handler(int signo) {
119     int retcode;
120     if (signo == SIGTERM) {
121         LOG_TRACE(LOG_NOTICE, "Received SIGTERM. Server is cleaning up shared
122             memory and is going to close...");
123         retcode = joiningAllPThreads(myPThreadStruct);
124         handle_error(retcode, "Could not joining all PThreads", PROCESS_EXIT);
125         if (retcode == 0)
126             LOG_TRACE(LOG_NOTICE, "Successfully joined all Client PThreads");
127         cleanup(shm_id);
128         LOG_TRACE(LOG_NOTICE, "Bye bye . . .");
129         exit(1);
130     } else {
131         LOG_TRACE(LOG_NOTICE, "Received other than . Server is cleaning up shared
132             memory and is going to close...");
133         retcode = joiningAllPThreads(myPThreadStruct);
134         handle_error(retcode, "Could not joining all PThreads", PROCESS_EXIT);
135         if (retcode == 0)
136             LOG_TRACE(LOG_NOTICE, "Successfully joined all Client PThreads");
137         cleanup(shm_id);
138         LOG_TRACE(LOG_NOTICE, "Bye bye . . .");
139         exit(1);
140     }
141 }
142
143 /* initializing Shared Memory Control Set*/
144 int initshm(char *shm_start) {
145     LOG_TRACE(LOG_INFORMATIONAL, "Creating shm Control Set ...");
146     shm_ctr = malloc(sizeof(struct shm_ctr_struct));
147     shm_ctr->shm_size = TOT_SHM_SIZE;
148     shm_ctr->isfree = TRUE;
149     shm_ctr->isLast = TRUE;
150     shm_ctr->next = shm_ctr;
151     shm_ctr->prev = shm_ctr;

```

```

150 shm_ctr->filename = "NULL";
151 shm_ctr->filedata = shm_start;
152 LOG_TRACE(LOG_INFORMATIONAL, "... Done\n");
153 LOG_TRACE(LOG_INFORMATIONAL, "Shared Memory ID = %i\n", shm_id);
154 LOG_TRACE(LOG_INFORMATIONAL, "Filename is: %s\n", shm_ctr->filename)
    ;
155 LOG_TRACE(LOG_INFORMATIONAL, "Shared Memory Start location = %p\n",
    &(shm_ctr->filedata));
156 return TRUE;
157 }
158
159 int main(int argc, char *argv[]) {
160     int retcode;
161     LOG_TRACE(LOG_INFORMATIONAL, "\nServer started. Is now initializing the
        setup...\n");
162
163     signal(SIGINT, my_handler);
164
165     LOG_TRACE(LOG_INFORMATIONAL, "Setting up the valid arguments...");
166     setValidServerArguments(); //setting up all valid arguments
167     LOG_TRACE(LOG_INFORMATIONAL, "... Done\n");
168
169     retcode = setup_shm();
170     handle_error(retcode, "Shared Memory could not be created.\n", PROCESS_EXIT);
171
172     char *shm_start = shmat(shm_id, NULL, 0);
173     LOG_TRACE(LOG_INFORMATIONAL, "... Done\n");
174
175     /* init shared memory control*/
176     retcode = initshm(shm_start);
177     handle_error(retcode, "Could not create Shared Memory Control Set...\n",
        PROCESS_EXIT);
178
179     /* init struct for PThread handling Clients */
180     myPThreadStruct = (struct pthread_struct *) malloc(sizeof(struct pthread_struct));
181     myPThreadStruct->isLast = 1;
182     myPThreadStruct->nextClient = myPThreadStruct;
183
184     /* if no arguments is chosen, output the usage of the Server */
185     if (argc == 1) {
186         usage();

```

```

187 }
188 /* if arguments are chosen, validate the arguments */
189 else {
190     printf("Verify valid arguments ...\n");
191     initValidServerArguments(argc, argv);
192 }
193 /* if no port for the server was chosen, set it to default port = 7000 */
194 if (validArguments[1].isSet == 0) {
195     printf("There was no argument for the Server-Port. It will no be set to default =
196         7000\n");
197     ServerPort = 7000;
198     validArguments[1].isSet = 1;
199 }
200
201 retcode = setTCPServer();
202 handle_error(retcode, "TCP Server settings could not be established!\n",
203     PROCESS_EXIT);
204 ServerListen();
205
206 /* clean up shared memory */
207 cleanup(shm_id);
208 }
209
210 int setTCPServer() {
211     int retcode;
212     printf("Set up TCP-Server settings ...\n");
213     //printf("LOGLEVEL_DEBUG = %i", LOGLEVEL_DEBUG);
214
215     /* Create socket for incoming connections */
216     servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
217     handle_error(servSock, "socket() failed", PROCESS_EXIT);
218
219     /* Construct local address structure */
220     memset(&squareServAddr, 0, sizeof(squareServAddr));
221     /* Zero out structure */
222     squareServAddr.sin_family = AF_INET; /* Internet address family */
223     squareServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming
224         interface */
225     squareServAddr.sin_port = htons(ServerPort); /* Local port */
226
227     /* Bind to the local address */

```

```

225 retcode = bind(servSock, (struct sockaddr *) &squareServAddr, sizeof(squareServAddr
    ));
226 handle_error(retcode, "bind() failed", PROCESS_EXIT);
227
228 /* Mark the socket so it will listen for incoming connections */
229 retcode = listen(servSock, MAXPENDING);
230 handle_error(retcode, "listen() failed", PROCESS_EXIT);
231
232 return 1;
233 }
234
235 void ServerListen() {
236     LOG_TRACE(LOG_INFORMATIONAL, "Server is now going to Listening Mode for
        Clients.");
237     LOG_TRACE(LOG_INFORMATIONAL, "Client can connect to Server on Port %i",
        ServerPort);
238     pthread_t _myPThread;
239     int threadcounter = 0;
240
241     /* Run forever */
242     while (TRUE) {
243
244         /* Set the size of the in-out parameter */
245         client_address_len = sizeof(ClientSocketAddress);
246
247         /* Wait for a client to connect */
248         LOG_TRACE(LOG_INFORMATIONAL, "Waiting for Client to connect...");
249         clientSocket = accept(servSock, (struct sockaddr *) &ClientSocketAddress, &
            client_address_len);
250         handle_error(clientSocket, "accept() failed", 0);
251
252         /* fill struct for pthread */
253         struct client_param_struct cps;
254         cps.clientSocket = clientSocket;
255
256         /*handle the client and create per client a single thread */
257         pthread_create(&_myPThread, NULL, &handle_tcp_client, &cps);
258         addPThread(myPThreadStruct, _myPThread, threadcounter);
259
260         /* clntSock is connected to a client ! */

```

```

261     LOG_TRACE(LOG_WARNING, "Handling Client %s", inet_ntoa(
        ClientSocketAddress.sin_addr));
262
263 }
264 /* NOT REACHED: */
265 exit(0);
266 }
267
268 void runClientCommand(char *recMessage[], char *command, int clntSocket, int
    thread_count) {
269     //LOG_TRACE(LOG_INFORMATIONAL, "Command from Client was: %s",
        command);
270     char *sendtoClient = (char *) malloc(MAX_FILE_LENGTH);
271     memset(sendtoClient, '\0', sizeof(sendtoClient));
272
273     /* if Client want to exit, join PThread and Exit */
274     if (strcmp(command, "EXIT") == 0) {
275         LOG_TRACE(LOG_INFORMATIONAL, "Received EXIT from a Client");
276         LOG_TRACE(LOG_INFORMATIONAL, "T%i: PThread is now going to exit",
            thread_count);
277         pthread_exit(NULL);
278     }
279
280
281     /* CREATE command */
282     if (strcmp(command, "CREATE") == 0) {
283
284         LOG_TRACE(LOG_INFORMATIONAL, "Will no try to create a new file...");
285         //printf("rec Message = %s\n", *recMessage);
286
287         char *tmpcontent = (char *) malloc(MAX_FILE_LENGTH);
288         tmpcontent = getFileContent(recMessage);
289         char *filecontent = strdup(tmpcontent);
290         free(tmpcontent);
291
292         char *filename = strdup(recMessage[2]);
293         LOG_TRACE(LOG_INFORMATIONAL, "Filesize = %i \t Content = %s", (int)
            strlen(filecontent), filecontent);
294
295         char *returnvalue = malloc(sizeof(char) * MAX_FILE_LENGTH);
296         returnvalue = writeNewFile(shm_ctr, filename, filecontent, strlen(filecontent));

```

```

297     if (returnvalue > 0) {
298         LOG_TRACE(LOG_INFORMATIONAL, "Sending message to Client: %s",
                returnvalue);
299         send(clntSocket, returnvalue, strlen(returnvalue), 0);
300     }
301     free(returnvalue);
302 }
303
304 /* Reading File */
305 else if (strcmp(command, "READ") == 0) {
306
307     char * returnvalue = readFile(shm_ctr, recMessage[2]);
308     LOG_TRACE(LOG_INFORMATIONAL, "READ Command: Sending message to
        Client: %s", returnvalue);
309     send(clntSocket, returnvalue, strlen(returnvalue), 0);
310 }
311
312 else if (strcmp(command, "LIST") == 0) {
313     sendtoClient = get_all_shm_blocks(shm_ctr);
314     send(clntSocket, sendtoClient, strlen(sendtoClient), 0);
315 }
316
317 else if (strcmp(command, "UPDATE") == 0) {
318     LOG_TRACE(LOG_INFORMATIONAL, "Will no try to UPDATE the file \"%i\"
        ", recMessage[2]);
319     int retcode;
320     retcode = checkifexists(shm_ctr, recMessage[2]);
321     /* if file does not exist, send message to Client */
322     if (!retcode) {
323         sendtoClient = getSingleString("File with the name \"%s\" does not exist!\n",
            recMessage[2]);
324         send(clntSocket, sendtoClient, strlen(sendtoClient), 0);
325     }
326     /* if file exist, delete it and create it new */
327     else {
328         retcode = deleteFile(shm_ctr, recMessage[2]);
329         /* if deleting was successful, call runClientCommand and */
330         if (retcode) {
331
332             /* combine now the free blocks */
333             retcode = combine(shm_ctr);

```

```

334      /* repeat until there is no more deviding option */
335      while (retcode == TRUE) {
336          retcode = combine(shm_ctr);
337      }
338
339      /* create the new file with the content */
340      char *tmpcontent = (char *) malloc(MAX_FILE_LENGTH);
341      tmpcontent = getFileContent(recMessage);
342      char *filecontent = strdup(tmpcontent);
343      free(tmpcontent);
344
345      char *filename = strdup(recMessage[2]);
346      LOG_TRACE(LOG_INFORMATIONAL, "Filesize = %i \t Content = %s", (
          int) strlen(filecontent), filecontent);
347
348      char *returnvalue = malloc(sizeof(char) * 256);
349      returnvalue = writeNewFile(shm_ctr, filename, filecontent, strlen(filecontent))
          ;
350      if (returnvalue > 0) {
351          LOG_TRACE(LOG_INFORMATIONAL, "Sending message to Client: %s",
              returnvalue);
352          send(clntSocket, returnvalue, strlen(returnvalue), 0);
353      }
354      free(returnvalue);
355
356  }
357
358  /*if deleting was not successful */
359  else {
360      LOG_TRACE(LOG_INFORMATIONAL, "Updating file \"%s\" was not
          successful", recMessage[2]);
361      sendtoClient = getSingleString("Updating file \"%s\" was not successful",
          recMessage[2]);
362      send(clntSocket, sendtoClient, strlen(sendtoClient), 0);
363  }
364
365  }
366
367  }
368
369  /* DELETE <filename>: DELETE Filename from memory */

```

```

370 else if (strcmp(command, "DELETE") == 0) {
371     printf("Client wants to DELETE a file.\n");
372     int retcode;
373     retcode = checkifexists(shm_ctr, recMessage[2]);
374
375     /* if file does not exist, send message to Client */
376     if (!retcode) {
377         sendtoClient = getSingleString("File with the name \"%s\" does not exist!\n",
378             recMessage[2]);
379         send(clntSocket, sendtoClient, strlen(sendtoClient), 0);
380     }
381     /* else delete the file */
382     else {
383         retcode = deleteFile(shm_ctr, recMessage[2]);
384
385         /* if deleting was successful */
386         if (retcode) {
387             sendtoClient = getSingleString("File with name \"%s\" was successfully deleted
388                 .", recMessage[2]);
389             send(clntSocket, sendtoClient, strlen(sendtoClient), 0);
390             printf("After deleting: SHM Block is now:\n");
391             printf(get_all_shm_blocks(shm_ctr));
392         }
393         LOG_TRACE(LOG_INFORMATIONAL, "Will now try to combine free blocks...
394             ");
395         retcode = combine(shm_ctr);
396         /* repeat until there is no more deviding option */
397         while (retcode == TRUE) {
398             printf("\n\n\n", retcode);
399             retcode = combine(shm_ctr);
400             printf(get_all_shm_blocks(shm_ctr));
401         }
402     }
403     /* if nothing compared */
404     else {
405         LOG_TRACE(LOG_DEBUG, "No match. Send nothing to commit to client");
406         send(clntSocket, "Nothing to commit", strlen("Nothing to commit"), 0);
407     }
408     free(sendtoClient);

```



```

408 }
409 }
410
411 void handle_tcp_client(void* parameters) {
412     int istrue = 1;
413     LOG_TRACE(LOG_INFORMATIONAL, "New PThread created for Client.\t ID =
        %u", (unsigned int) pthread_self());
414     /* Cast the given parameter back to int ClnSocket */
415     struct client_param_struct* p = (struct client_param_struct*) parameters;
416     int thread_count = p->thread_count;
417     int clntSocket = p->clientSocket;
418     LOG_TRACE(LOG_INFORMATIONAL, "clientSocket = %i", clntSocket);
419     int retcode;
420     char recBuffer[MAXRECWORDS]; /* Buffer for string */
421     int recvMsgSize; /* Size of received message */
422     /* array to save the single words of the received message */
423     char *recMessage[MAXRECWORDS];
424     while (istrue) {
425
426         /* reset Buffer for next transmission */
427         memset(recBuffer, 0, sizeof(recBuffer));
428         memset(recMessage, 0, sizeof(recMessage));
429
430         /* Receive message from client */
431         LOG_TRACE(LOG_INFORMATIONAL, "Waiting for reveicing message from
            Client.");
432         recvMsgSize = recv(clntSocket, recBuffer, BUFSIZE - 1, 0);
433         handle_error(recvMsgSize, "recv() failed", NO_EXIT);
434         if (recvMsgSize == 0) {
435             break;
436         }
437         LOG_TRACE(LOG_INFORMATIONAL, "Received message from Client %s: %s",
            inet_ntoa(ClientSocketAddress.sin_addr), recBuffer);
438         breakCharArrayInWords(recMessage, recBuffer);
439         /* check is effective message is equal to expected message size */
440         int effLength = (int) atoi(recMessage[0]);
441         if (effLength == recvMsgSize) {
442             /* check if 1st word of message is a valid command */
443             retcode = getValidServerCommand(recMessage[1]);
444
445             /* if command is valid */

```

```

446     if (retcode) {
447         LOG_TRACE(LOG_INFORMATIONAL, "It is a valid command: %s",
448             recMessage[1]);
449         runClientCommand(recMessage, recMessage[1], clntSocket, thread_count);
450     };
451     }
452     recBuffer[recvMsgSize] = '\000'; // set End Termination at the end of the Buffer
453 }
454 close(clntSocket); /* Close client socket */
455 }
456 void breakCharArrayInWords(char *recMessage[], char *recBuffer[]) {
457     /* break now the received Message into a string array where the sign " " breaks
458        words */
459     char breaksign[] = " ";
460     char *token = malloc(sizeof(char) * MAX_FILE_LENGTH);
461     int count = 0;
462     /* get the first token */
463     token = strtok(recBuffer, breaksign);
464     recMessage[count] = token;
465     count++;
466     /* walk through other tokens */
467     while (token != NULL) {
468         token = strtok(NULL, breaksign);
469         recMessage[count] = malloc(sizeof(char) * MAX_WORD_SIZE);
470         recMessage[count] = token;
471         count++;
472     }
473 }
474 int setLogLevel(int logLevel) {
475     #ifndef DEBUG
476     #define DEBUG
477     #endif
478     LOGLEVEL = logLevel;
479     return 1;
480 }

```

Listing A.1: Server.c

A.2.2. Client.c

```

1  /*
2  * File:    Client.c
3  * Author:  Micha Schönberger
4  * Modul:   Concurrent Programming in C
5  *
6  * Created: 14.04.2014
7  * Project:  https://github.com/schoenml/concurrent_c.git
8  */
9
10 /* ----- How to user this Program
11  -----
12  1) ...
13  2) ...
14  3) ...
15
16 -----
17
18 */
19
20 #include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
21 #include <errno.h>
22 #include <netinet/in.h>
23 #include <netinet/tcp.h>
24 #include <stdio.h>
25 #include <stdio.h> /* for printf() and fprintf() and ... */
26 #include <stdlib.h> /* for atoi() and exit() and ... */
27 #include <string.h> /* for memset() and ... */
28 #include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
29 #include <time.h>
30 #include <sys/types.h>
31 #include <unistd.h> /* for close() */
32 #include <signal.h>
33 #include <itskylib.h>
34 #include <unistd.h>
35
36 #define BUFSIZE 65535 /* Size of receive buffer */
37 #define SENDBUFFER 8192 /* Size of receive buffer */
38
39 int sock; /* Socket descriptor */
40
41 struct sockaddr_in server_address; /* Square server address */

```

```

38 unsigned short server_port; /* Square server port */
39 char *server_ip; /* Server IP address (dotted quad) */
40
41 unsigned short int send_length;
42
43 /* forward declarations of functions */
44 void usage(const char *argv0, const char *msg);
45 void my_handler(int signo);
46 void clearBuffers(char sendbuffer[BUFSIZE], char recbuffer[BUFSIZE]);
47 void calcMsgToSend(char sendbuffer[BUFSIZE], char tmpsquare_buffer[BUFSIZE]);
48 void closeSocket();
49
50 /*
51  * BEGIN OF Client.c
52  */
53
54 /* shows the usage of the Client to connect to the Server */
55 void usage(const char *argv0, const char *msg) {
56     if (msg != NULL && strlen(msg) > 0) {
57         printf("%s\n\n", msg);
58     }
59     printf("Usage\n\n");
60     printf("%s <Server IP> <number> [<Port>]\n", argv0);
61     exit(1);
62 }
63
64 /* Handles the Signals which are received by the Client */
65 void my_handler(int signo) {
66     if (signo == SIGTERM) {
67         printf("Received and ignored SIGTERM.\n");
68     } else if (signo == SIGINT) {
69         printf("Received Ctrl-C. Send now Command to Server that Client exit.\n");
70         char sendbuffer[BUFSIZE];
71         calcMsgToSend(sendbuffer, "EXIT");
72         send(sock, sendbuffer, strlen(sendbuffer), 0);
73         usleep(100);
74         exit(1);
75     } else {
76         printf("unknow Signal %d will be ignored\n", signo);
77     }
78 }

```

```

79
80 int main(int argc, char *argv[]) {
81     signal(SIGINT, my_handler);
82     int retcode;
83
84     if (is_help_requested(argc, argv)) {
85         usage(argv[0], "");
86     }
87
88     char recbuffer[BUFSIZE]; /* Buffer for string */
89     char sendbuffer[BUFSIZE]; /* Buffer for string */
90
91     if (argc < 2 || argc > 3) { /* Test for correct number of arguments */
92         usage(argv[0], "wrong number of arguments");
93     }
94
95     server_ip = argv[1]; /* First arg: server IP address (dotted quad) */
96
97     if (argc == 3) {
98         server_port = atoi(argv[2]); /* Use given port, if any */
99     } else {
100         server_port = 7000; /* 7000 is a free port */
101     }
102
103     /* Create a reliable , stream socket using TCP */
104     sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
105     handle_error(sock, "socket() failed", PROCESS_EXIT);
106
107     /* Construct the server address structure */
108     memset(&server_address, 0, sizeof(server_address));
109     /* Zero out structure */
110     server_address.sin_family = AF_INET; /* Internet address family */
111     server_address.sin_addr.s_addr = inet_addr(server_ip); /* Server IP address */
112     server_address.sin_port = htons(server_port); /* Server port: htons host to network
        byte order */
113
114     /* Establish the connection to the square server */
115     retcode = connect(sock, (struct sockaddr *) &server_address, sizeof(server_address));
116     handle_error(retcode, "connect() to Server failed", PROCESS_EXIT);
117
118     /* Run forever */

```

```

119
120 while (TRUE) {
121     int num;
122     char tmpsquare_buffer[BUFSIZE];
123     printf("\n# Enter Command for Server: ");
124     fgets(tmpsquare_buffer, BUFSIZE - 2, stdin);
125     strcat(tmpsquare_buffer, " ");
126     clearBuffers(sendbuffer, recbuffer);
127     calcMsgToSend(sendbuffer, tmpsquare_buffer);
128
129     if (strlen(sendbuffer) >= SENDBUFFER - 20) {
130         printf("The message to send is too long. Use a shorter text!\n");
131
132     } else {
133         if ((send(sock, sendbuffer, strlen(sendbuffer), 0)) == -1) {
134             fprintf(stderr, "Failure Sending Message\n");
135
136         } else {
137             printf("# Message being sent: %s\n", sendbuffer);
138             recbuffer[0] = '\0';
139             recv(sock, recbuffer, sizeof(recbuffer), 0);
140
141             printf("# Message from Server: %s\n", recbuffer);
142         }
143     }
144 }
145 closeSocket();
146 }
147
148 /* clear the recbuffer and the sendbuffer */
149 void clearBuffers(char sendbuffer[BUFSIZE], char recbuffer[BUFSIZE]) {
150     /* check length of buffer and add int before it. This is the control for the server
151        to check if he received the complete message */
152     /* clears the buffer for next transmission */
153     memset(recbuffer, 0, strlen(recbuffer));
154     memset(sendbuffer, 0, strlen(sendbuffer));
155     recbuffer[0] = '\0';
156 }
157
158 /* here the message will be calculated. This is a short transmission protocol to
159    ensure that all traffic was sent */

```

```
158 void calcMsgToSend(char sendbuffer[BUFSIZE], char tmpsquare_buffer[BUFSIZE]) {
159     char intlen [10];
160     /* 8 for int +1 for space, +1 for \0 */
161     sprintf(intlen, "%9d", strlen(tmpsquare_buffer)+8+1 +1);
162
163     /* the message to send is: <length of following message><message> e.g. 5abcd (the
        '0' char will be added to <length of msg> */
164     strcat(sendbuffer, intlen);
165     strcat(sendbuffer, " ");
166     strcat(sendbuffer, tmpsquare_buffer);
167 }
168
169 void closeSocket() {
170     close(sock);
171 }
```

Listing A.2: Client.c

Literaturverzeichnis

- [1] *Apache Lucene - Index File Formats.* http://lucene.apache.org/core/3_5_0/fileformats.html, may 2014.
- [2] *Lucene Query Parser Syntax.* http://lucene.apache.org/core/2_9_4/queryparsersyntax.html, may 2014.
- [3] *Who is using Lucene/Solr.* <http://searchhub.org/2012/01/21/who-uses-lucenesolr/>, may 2014.
- [4] MCCANDLESS, MICHAEL; HATCHER, ERIK; GOSPONDENETIC OTIS: *Lucene in Action - Second Edition.* Manning Publications Co, 2010.