

Concurrent Programming in C

Seminararbeit FS 2014

**Dozent**: Nico Schottelius  
xsou@zhaw.ch

**Student:** Micha Schönenberger  
schoenm1@students.zhaw.ch

**Inhaltsverzeichnis**

1 Versionierung 4

2 Aufwände 5

3 Einleitung 8

3.1 Rahmenbedingungen 8

3.1.1 Termine 8

3.1.2 Administratives 8

3.1.3 Abgabebedingungen 8

3.1.4 Vortrag / Präsentation 8

3.1.5 Lernziele 9

3.1.6 Lerninhalte 9

3.2 Das Projekt 9

3.3 Ausgangslage 9

4 Anleitung zur Nutzung 11

5 Umsetzung des Projektes 12

5.1 Voraussetzungen 12

5.2 Libraries 12

5.3 Programmierumgebung 12

5.4 LOG/DEBUG 13

5.5 Speicherverwaltung – Buddy System 14

5.5.1 Wie wird das komplette System gemanaged? 14

5.5.2 Wie finde ich für das File den optimalen Block im Shared Memory? 14

5.5.3 Gibt es nur zu grosse Blöcke, wie werden die geteilt? 15

5.6 Locks 17

5.7 Files und deren Funktionen 18

6 Literaturverzeichnis 20

7 Anhang 21

7.1 Screenshots zum Server 21

7.1.1 Shared Memory vor dem Einfügen eines neuen Files 21

7.1.2 Auteilen des Shared Memory in der Funktion devide() 21

7.1.3 Shared Memory nach dem Einfügen eines neuen Files 22

# Versionierung

|  |  |  |
| --- | --- | --- |
| **Version** | **Datum** | **Bemerkung** |
| V0.1 | 15.03.2014 | Ersterstellung |
| 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 | 03.05.2014 | TCP Connection Protocol, Loglevel |
| V0.9 | 06.05.2014 | CREATE, LIST, LOG, Fehlebehebungen |
| V1.0 | 27.05.2014 | One Thread per Client, Refactoring, Commenting Code |
| V1.1 | 01.06.2014 |  |
| V1.2 | 02.06.2014 | Implementing RWLock for Creating File |

# Aufwände

|  |  |  |
| --- | --- | --- |
| **Datum** | **Zeitaufwand** | **Bezeichnung** |
| 15.03.2014 | 3.75h | * Ersterstellung Dokumentation * Github Repo erstellen * Einlesen Buch Kapitel 15 (Semaphore, Shared Memory…) * Erstellen Debian VM * SSH Key-gen for Github (Mac and Debian) |
| 17.03.2014 | 0.5h | * Dokumentation: Einleitung (Rahmenbedingungen, Projekt, Ausgangslage) |
| 07.04.2014 | 1.75 h | * Grundgerüst erstellen, LOG-LEVEL definieren |
| 08.04.2014 | 3 h | * Parsing Argumente bei Programmstart Log-Level Implementierung |
| 13.04.2014 | 3.5 h | * 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“ . ->Mail an Nico Schottelius |
| 15.04.2014 | 1h | * Installing eUML -> not working, just with .java |
| 01.05.2014 | 9 h | * Degub 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. Nach etlichen Stunden Aufgabe   Fazit Arbeiten Heute:   * 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 erst beim Senden Client): Die Länge der zu sendenden Zeichen werden gezählt und diese Zahl mitgesendet. Der Server splittet den Stream in Arrays und überprüft, ob die empfangenen Daten gleich viele Zeichen aufweisen wie die zu erwartende. |
| 04.05.2014 | 6h | * Überprüfung 1. Wort von Client als Command-Argument (Momentan nur Create File) |
| 05.05.2014 | 5.75h | * Verfeinern CREATE Command * LOG verbessern |
| 05.05.2014 | 8h | * beim CREATE vom 2. File wurde der Name des 1. 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 Adressen 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 trotzdem noch * DELETE und READ fertig implementieren (ohne Lock) |
| 26.05.2014 | 1.75h | * Erstellen von PThreads für Clients |
| 27.06.2014 | 4.25h | * Erstellen von PThreads für Client * Implementierung ReadWrite Lock mit pthread\_rwlock\_t (Momentan nur ReadLock beim Lesen) * Kommentieren von Code * löschen von altem, nicht mehr benutztem Code * Anpassen Log-Design (damit besser lesbar) |
| 28.06.2014 | 2.0h | * Update Dokumentation * Anpassen Version Github/Dokumentation * Kapitel 5.7 beginnen |
| 31.05.2014 | noch nachschauen im excel | * Fehlerbehebung PThreadList * Senden von EXIT bei Beenden von Client an Server * Joining PThread nach Client-EXIT bei Server |
| 01.06.2014 | 7.0h | * Joining PThread nach Client-EXIT bei Server fertig * Probleme Segmentation Fault beim löschen des letzten Files (mehrere Stunden suchen) -> Problem war Test.txt (fix in Code als Testfile) * Implementierung von RW-Lock bei DELETE File |
| 02.06.2014 | xx h | * Fehlerbehebung bei Übermittlung von grösseren Fileinhalten * Code kommentieren * Dokumentation erweitern |
|  |  |  |

# Einleitung

## Rahmenbedingungen

Die Aufgabenstellung und die Rahmenbedingungen wurden über Github (https://github.com/telmich/zhaw\_seminar\_concurrent\_c\_programming) veröffentlicht.

Anbei ein Auszug aus den wichtisten Eckdaten und Anforderungen:

### Termine

|  |  |
| --- | --- |
| **Datum** | **Bezeichnung** |
| 13.03.2014 | Kick-off Meeting |
| 16.03.2016 | Angabe des Git-Repository |
| 22.06.2014 | Abgabe 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 |

### 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.

### 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

### Vortrag / Präsentation

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

### Lernziele

* Die Besucher des Seminars verstehen was Concurrency bedeutet und welche Probleme und Lösungssansätze 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.

### 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).

## 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

## 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 0 (Aufwände) die Zeiten erfasst und ausgewiesen.

# Anleitung zur Nutzung

HIER FEHLT NOCH DIE ANLEITUNG

# Umsetzung des Projektes

## 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.

## Libraries

Im Unterricht des Modules „Systemsoftware“ wurden verschiedene Libraries durch den Dozenten zur Verfügung gestellt.  
Diese sollen, da sie einige Grundfunktionen wir das Error-Handling bereits beinhalten, in diesem Projekt ebenfalls genutzt werden. Die so genutzten Dateien werden nicht explizit als Quelle erwähnt. Sie besitzten jedoch im Kopf die Daten des Dozenten und sind als exterene Datei erkennbar.

Beispiel:

/\* (C) IT Sky Consulting GmbH 2014

\* http://www.it-sky-consulting.com/

\* Author: Karl Brodowsky

\* Date: 2014-02-27

\* License: GPL v2 (See https://de.wikipedia.org/wiki/GNU\_General\_Public\_License )

\*

\* This file is inspired by

\* http://cs.baylor.edu/~donahoo/practical/CSockets/code/HandleTCPClient.c

\*/

## 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. Zugegriffen 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 Untericht 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

## LOG/DEBUG

Die Impelementierung 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 anlog zu den syslog LOG-Level erstellt:

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

## 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

### Wie wird das komplette System gemanaged?

Hierzu wurde ein Struct erstellt, welches das Shared Memory kontrollieren soll.

struct shm\_ctr\_struct {  
 int shm\_size; //size of shm-block  
 int isfree; //indicates if block is free or not  
 int isLast; //indicates the end of shared memory  
 struct shm\_ctr\_struct \*next;  
 struct shm\_ctr\_struct \*prev;  
 char \*filename;  
 char \*filedata; // just this pointer is a pointer to shm  
};

|  |  |
| --- | --- |
| **Struct Attribut** | **Bezeichnung** |
| int shm\_size | Grösse des Blockes des Shared Memory Bereiches |
| int isfree | TRUE wenn Block frei ist, FALSE wenn Block besetzt ist |
| int isLast | TRUE wenn es der letzte Block ist, sonst FALSE |
| struct shm\_ctr\_struct \*next | Pointer auf den nächsten Block (zeigt auf sich selber, wenn es der letzte Block ist) |
| struct shm\_ctr\_struct \*next | Pointer auf den vorhergehenden Block (zeigt auf sich selber, wenn es der erste Block ist) |
| char \*filename | Pointer auf den Filenamen, der im Block gespeichert ist (NULL wenn kein File gespeichert ist) |
| char \*filedate | 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. |

### Wie finde ich für das File den optimalen Block im Shared Memory?

Hierzu wurde die Funktion **find\_shm\_place(…)** erstellt.

Diese Funktion beginnt beim ersten Eintrag des Structes **shm\_ctr\_struct** und sucht über alle vorhanden Blöcke (über den next-Pointer) einen optimalen Block.

Optimal bedeutet, dass er grösser oder gleich der Grösse des Files sein muss, aber nicht grösser als das doppelte (da Speicherplatzverschwendung)

Zusätzlich muss er frei sein (isfree = TRUE).

### 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 = 65536 |

Ist die Filegrösse = 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 |

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, welcher frei ist, 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 7.1.2 zu finden. Ebenfalls im Anhang 7.1.1 und 7.1.3 ist das Shared Memory vor und nach dem Einfügen eines neuen Files zu sehen.

## 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.

Somit wurde auf pthread\_rwlock\_t zurückgegrifen.

Die Impelementation 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 ein File gelockt werden, kann das elegant gelöst werden:

* Wenn das File gelesen werden möchte, muss zwingend die Adresse des enstprechenden Strukt bereits vorhanden sein = struct shm\_ctr\_struct \*shm\_ctr
* Nun kann ein ReadLock über  
  pthread\_rwlock\_rdlock(&(shm\_ctr->rwlockFile));  
  gemacht werden.

## Files und deren Funktionen

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

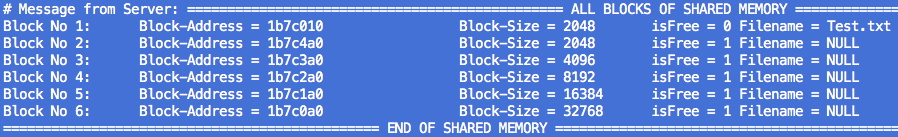
|  |  |
| --- | --- |
| **Filename** | **Funktion** |
| **myfunctions.c**  beinhaltet eigene definierte Funktionen | **power(int x, int y)** gibt als return Wert x^y  **getFixCharLen(char \*mychar, int mylength)** füllt einen CharPointer bis zur gewählten Länge auf. Wird benötigt für schöne Darstellung 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 |
| **shm\_control.ch**  Hier wird das Shared Memory kontrolliert | **struct shm\_ctr\_struct\* find\_shm\_place(struct shm\_ctr\_struct \*shm\_ctr, int filesize)** sucht einen geeigneten Platz im SHM für ein neues File mit der Filegrösse ‚filesize‘ @return: struct shm\_ctr\_struct = FALSE (=0) wenn kein guter Platz gefunden, sonst Adresse des Structs  **int devide(struct shm\_ctr\_struct \*shm\_ctr, int untilSize)** teilt das SHM solange auf (rekursiv), bis ein geeigneter Block vorhanden ist für das neue File @return: TRUE wenn erfolgreich, FALSE wenn nicht erfolgreich  **int combine(struct shm\_ctr\_struct \*shm\_ctr)** nach dem löschen eines Files wird hier überprüft, ob zwei freie Blöcke wieder miteinander verbunden werden können. So wird vermieden, dass viele kleine freie Blöcke sich ansammeln @return: TRUE wenn erfolgreich, FALSE wenn nicht erfolgreich |
| **valid-args.h** | **int getValidServerCommand(char \*command)** @return: TRUE wenn erfolgreich, FALSE wenn nicht erfolgreich **void setValidServerArguments()** setzen der gültigen Serverargumente beim Start des Servers  **int setServerPort(int myserverPort)** setzt den Port des Server, auf welchem er auf Clients hört @return: TRUE wenn erfolgreich, FALSE wenn nicht erfolgreich  **void usage()** Wird angezeigt, wenn keine/falsche Argumente gewählt werden  **void initValidServerArguments(int argc, char \*argv[])** Initialisiert die durch den Start des Server gegebenen Argumente und prüft sie, ob sie gültig sind  **void setValidServerArguments()** setzt die durch die Argumente bei Start des Servers gegebenen Werte als Parameter für den Server |

# Literaturverzeichnis

# Anhang

## Screenshots zum Server

### Shared Memory vor dem Einfügen eines neuen Files



### Auteilen des Shared Memory in der Funktion devide()



### Shared Memory nach dem Einfügen eines neuen Files

