Seminararbeit
Concurrent C Programming

Fileserver

Zürcher Hochschule für
Angewandte Wissenschaften
School of Engineering

:
Max Schrimpf

Betreuungsperson:
Nico Schottelius

Frühlingssemester 2014

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Ausgangslage . . . 2
1.2 Ziele der Arbeit . . . 2
1.3 Aufgabenstellung . . . 2
1.4 Erwartete Resultate . . . 3

2 Anleitung zur Nutzung **3**
2.1 Make . . . 3
2.2 Server *(run)* . . . 4
2.3 Modultest *(test)* . . . 5
2.4 Client *(client)* . . . 6

3 Lösungsstrategie **7**
3.1 Weg . . . 7
3.2 Probleme und deren Lösungen . . . 7
3.2.1 Erweiterung des Protokolls . . . 7
3.2.2 Concurrency . . . 8
3.2.3 Input parsing . . . 10
3.3 Test . . . 12
3.4 Mögliche weitere Schritte . . . 12

4 Fazit **13**

A Anhang **14**
A.1 Protokoll [3] . . . 14
A.2 Abbildungsverzeichnis . . . 15
A.3 Listings . . . 15
A.4 Literaturverzeichnis . . . 16

1 Einleitung[1]

1.1 Ausgangslage

Die Programmiersprache C ist im Vergleich zu anderen heutzutage verbreiteten Hochsprachen enorm systemnah und performant. Sie wird daher zum Beispiel in Bereichen wie der Systemprogrammierung, unter Anderem auch zur Implementation des Linux Kernels, eingesetzt. Durch die hiermit verbundenen Freiheiten und Vorteile kommen jedoch verschiedene Stolpersteine und diverse Möglichkeiten, sei es durch falschen Umgang mit Systemressourcen wie dem Arbeitsspeicher oder Anderem, das System nachhaltig zu beschädigen.

1.2 Ziele der Arbeit

Durch das Seminar „Concurrent C Programming“ soll den Studenten[2] der Umgang mit Concurrency Problemen in C, explizit bei der gemeinsamen Nutzung des Hauptspeichers, näher gebracht werden.

In dieser Seminararbeit soll ein virtueller File-Server in C programmiert werden. Es soll möglich sein mittels eines gegebenen Protokolls die Aktionen „CREATE“, „READ“, „UPDATE“ und „DELETE“ auf einem File auszuführen. Zusätzlich soll mit „LIST“ eine Liste aller derzeit bestehenden Files angezeigt werden können.

1.3 Aufgabenstellung

- Einarbeiten in das Seminarthema
- Konzeption und Entwicklung eines Programms, das gleichzeitig auf einen Speicherbereich zugreift
- Realisierung der Kommunikation mit einem gegebenen Protokoll (A.1) über TCP/IP oder Unix Domain Sockets
- Erstellen eines Seminarberichts
- Präsentation vor Kommilitonen

Die Implementation hat mithilfe von Threads oder Forks und Shared Memory (SHM) zu erfolgen. Die erstellten Dateien sind nur im Speicher vorhanden, das echte Dateisystem darf nicht genutzt werden. Es dürfen keine globalen Locks benutzt werden sondern es müssen mehrere gleichzeitige Clients mit jeweiligem Lock auf Dateiebene möglich sein.

Die genaue Definition des weiteren Funktionsumfangs ist dem Studenten überlassen.

[1] Vergleiche Aufgabenstellung im EBS

[2] Sämtliche personenbezogenen Bezeichnungen in der Aufgabenstellung müssen aufgrund der Abwesenheit weiblicher Studierender im Seminar nicht als geschlechtsneutral zu verstanden werden.

1.4 Erwartete Resultate

- Lauffähiger File-Server in C
- Test-Client mit Concurrent-Tests in beliebiger Programmiersprache
- Dokumentation zum Projekt

2 Anleitung zur Nutzung

2.1 Make

Es existieren folgende make Ziele:

all	erstellt die Hauptdateien der Seminararbeit: *run* und *test*
run	erstellt den Server: *run*
test	erstellt ein Executable für automatisierte Modultests: *test*
client	erstellt einen interaktiven Client zum manuellen Test des Servers: *client*
ragel	compiliert die Ragel (3.2.3) Datei *lib/messageProcessing.rl* in eine C Datei. Hierzu wird eine installierte Version des Ragel State Machine Compiler [4] benötigt. Dieser Schritt muss vom Dozenten nicht ausgeführt werden, da dem Projekt bereits eine Version von *lib/messageProcessing.c* beiliegen sollte.

2.2 Server *(run)*

Der Server erfüllt sämtliche in der Aufgabenstellung geforderten Funktionen. Um undefiniertes Verhalten zu verhindern wurden jedoch in der Kommunikation mit Clients folgende Einschränkungen vorgenommen:

- FILENAME und CONTENT von Files dürfen jeweils maximal 1024 Zeichen lang sein
- Die LENGTH kann daher in ihrer dezimalen Repräsentation nur 4 Zeichen lang sein

Diese Längen sind als Precompiler Statements in *lib/termPaperLib.h* hinterlegt und können somit, der mangels anderweitiger Anforderungen, nur mit einem Recompile verändert werden.

```
1  $ ./run -h
2  Help:
3
4  Usage:
5  ./run  [-p Port] [-d Out] [-i Out] [-e Out]
6
7  Server for the term paper in concurrent C programming
8  Will start a virtual file server that accepts connections via
9  TCP
10
11
12 [-p Port] Optional: Tries to connect to a server on the given port.
13             Default: 7000
14
15 [-d Loglevel] Optional: Alter the output for DEBUG messages.
16                 Default: No logging
17
18 [-i Loglevel] Optional: Alter the output for INFO messages.
19                 Default: stdout
20
21 [-i Loglevel] Optional: Alter the output for ERROR messages.
22                 Default: stderr
23
24  Possible log Outputs: 0 = No logging
25                        1 = Logfile
26                        2 = stdout
27                        3 = stderr
28
29
30 (c) Max Schrimpf - ZHAW 2014
```

Listing 1: Held Text des Servers

2.3 Modultest *(test)*

Der automatisierten Modultest überprüft die vom Server geforderten Funktionalitäten.

```
$ ./test -h
Help:

Usage:
./test  [-a IP] [-p Port] [-d Out] [-i Out] [-e Out]

Executes various tests on the fileserver
A running server at the given address is needed


[-a IP] Optional: Tries to connect to the server on the given IP.
         Default: 127.0.0.1

[-p Port] Optional: Tries to connect to a server on the given port.
           Default: 7000

[-d Loglevel] Optional: Alter the output for DEBUG messages.
               Default: No logging

[-i Loglevel] Optional: Alter the output for INFO messages.
               Default: stdout

[-i Loglevel] Optional: Alter the output for ERROR messages.
               Default: stderr

 Possible log Outputs: 0 = No logging
                       1 = Logfile
                       2 = stdout
                       3 = stderr


(c) Max Schrimpf - ZHAW 2014
```

Listing 2: Held Text des Modultests

2.4 Client *(client)*

Der Client stellt eine Möglichkeit dar Befehle auf dem Server via Commandline Interface auszuführen.

```
$ ./client -h
Help:

Usage:
./client [-c Line1] [-C Line2] [-a IP] [-p Port] [-d Out] [-i Out] [-e Out]

Connects to a server and provides an interactive multiline command
interface. You can simply start a new line with \n (Enter)
The command will be send to the server as soon as a empty line is
entered. The interactive interface can be ended using <Ctrl-C>
or by entering QUIT

You may also provide a command that is executed without an interactive
command interface using the [-c Line] option
If you want to specify a second line that is send to the server after
-c you can use -C


[-c Line1] Optional: A command that should be send to the server.
                     This option disables the interactive mode

[-C Line2] Optional: A second line for the command - only possible
                     if a first line is provided

[-a IP] Optional: Tries to connect to the server on the given IP.
          Default: 127.0.0.1

[-p Port] Optional: Tries to connect to a server on the given port.
           Default: 7000

[-d Loglevel] Optional: Alter the output for DEBUG messages.
                Default: No logging

[-i Loglevel] Optional: Alter the output for INFO messages.
                Default: stdout

[-i Loglevel] Optional: Alter the output for ERROR messages.
                Default: stderr

 Possible log Outputs: 0 = No logging
                       1 = Logfile
                       2 = stdout
                       3 = stderr


(c) Max Schrimpf - ZHAW 2014
```

Listing 3: Held Text des Clients

3 Lösungsstrategie

3.1 Weg

Das zeitgleich stattfindende Modul „Systemprogrammierung“ bot, dank verschiedener Hausaufgaben und Vorlagen [1], einen sehr guten Zugang zur Materie, wodurch der Einstieg ins Thema massiv erleichtert wurde. Unter Anderem behandelte genanntes Modul POSIX Threads und TCP/IP Kommunikation.

Die Seminararbeit basiert daher im Kern auf der Hausaufgabe Nr. 7 des Moduls, in der die Aufgabe war, einen Server für eine Kommunikation über TCP/IP mit beliebig vielen Clients zu realisieren.

Der hierzu gewählte Ansatz, auf dem Server für jede Verbindung einen Thread zur die Abwicklung des Clients zu erstellen, benötigte eine dynamische Liste an Threads, welche im Folgenden zur *lib/concurrentLinkedList.c* ausgebaut wurde.

Während der Realisierung der Seminararbeit wurde relativ schnell deutlich, dass gewisse häufig gebrauchte Funktionen zentralisiert werden sollten. Inspiriert durch die im genannten Modul zur Verfügung gestellte, GPLv2 lizenzierte Library, erfolgte die Zusammenfassung der notwendigen Funktionen in *lib/termPaperLib.c*. Die Library umfasst, unter Anderem, Logging, Errorhandling und Kommunikation via TCP/IP.

Für die Kapselung der Input Verarbeitung und Behandelung wurde schliesslich noch *lib/messageProcessing* erstellt. Sie enthält die zentrale Verarbeitung der Anfragen und die Interaktion mit der *lib/concurrentLinkedList.c.*

3.2 Probleme und deren Lösungen

3.2.1 Erweiterung des Protokolls

Gewisse Grenzfälle des Protokolls - beispielsweise bei einem mismatch von LENGTH und CONTENT - sind in der Beschreibung des Protokolls nicht behandelt. Für solche Situationen wurde eine kurze Input-Validierung vorgenommen:

1. Wenn die effektive Länge des CONTENT geringer ist, als in LENGTH angegeben, wird LENGTH durch die effektive Länge des CONTENT ersetzt.
   Hierdurch lässt sich verhindern, dass potentiell nicht initialisierte RAM Bereiche ausgelesen werden können etc.

2. Wenn die effektive Länge des CONTENT höher ist, als in LENGTH angegeben, wird der CONTENT auf die angegebene LENGTH gekürzt.

3. Die Maximalen Längen von CONTENT, FILENAME und LENGTH wurden definiert.
   Dabei soll verhindert werden, dass böswillig durch Anfragen riesige Mengen an Speicher belegt werden.

Um Probleme für die Gegenseite verständlicher zu machen, wurden weiterhin drei neue Antworten des Servers definiert:

- "COMMAND_UNKNOWN\n"
  Wenn die empfangene Nachricht keinem Befehl zugeordnet werden kann oder anderweitig nicht korrekt ist.

- "FILENAME_TO_LONG\n"
  Der Dateiname übersteigt das definierte Maximum.
- "CONTENT_TO_LONG\n"
  Der Dateiinhalt übersteigt das definierte Maximum.

3.2.2 Concurrency

Eine Kernanforderung der Aufgabenstellung war der Verzicht auf globale Locks. Um einen Reibungslosen Verlauf sicher zu stellen, muss dafür gesorgt werden, dass nicht z.B. ein UPDATE Request während des DELETE eines Files versucht, noch Änderungen vorzunehmen. Daher ist grundsätzlich ein Lock auf dem zu bearbeitenden File notwendig. Wenn jedoch die oben genannte Sequenz eintritt und sich der UPDATE Request in die Warteschlange des Locks einreiht besteht erneut ein Problem, da das DELETE den Lock für das File nicht entfernen kann und somit Überreste bestehen blieben.

Die für das Handling der Files gewählte Datenstruktur ist eine Linked List. Daher ist auch das zweite Problem relativ simpel dadurch lösbar, dass der jeweilige Vorgänger des zu bearbeitenden Files auch gelockt wird. Da am Vorgänger keine Änderungen vorgenommen werden, können sich nachfolgende Requests problemlos in diese Lock Warteschlange einreihen, ohne auf potentiell inkonsistente Daten zugreifen zu wollen (1).

Eine Optimierung dieses Ansatzes ist ein weiteres Lock auf dem Content, damit während Requests, welche die Sequenz der Files nicht ändern - also READ oder UPDATE - andere Requests, wenn sie z.B. ein File suchen und nicht auf den Inhalt des gerade benutzten File zugreifen wollen, nicht behindert werden .

Um die Integrität des Speichers sicher zu stellen, werden bei CREATE und UPDATE jeweils die zur Speicherung vorgesehenen Daten in einen eigens dafür allozierten Bereich kopiert, sodass der Aufrufer die bei ihm hinterlegten Daten nachfolgend problemlos verändern oder freigeben kann, ohne sie in der Liste zu verändern. Beim READ wird folglich auch eine eigens dafür allozierte Kopie der gespeicherten Daten zurückgegeben. Allerdings liegt hier ein grosser Stolperstein, da die zurückgegebene Kopie durch den grundsätzlichen Aufbau von C nur eine shallow copy ist. Dies bedeutet die Speicherbereiche, auf die die in der Kopie vorhandenen Pointer, zeigen werden nicht kopiert, weshalb beim Umgang mit ihnen besondere Vorsicht herrschen muss. Auch muss eine applikatorische Logik zur Freigabe der referenzierten Datenbereiche bestehen, da beim Löschen des Elementes der Linked List nur der vom Element selbst referenzierte Speicher, nicht aber der Speicher auf den etwaige Pointer zeigen, freigegeben wird.

Um einen sicheren Einstieg in die Liste, welche nicht zwingend Elemente enthalten muss, zu gewährleisten, existiert, neben den einzelnen Locks auf den Files, auch noch ein Lock auf der Grundstruktur der Liste. Letzteres kann durch keine Operation auf der Liste gelöscht werden und ist somit immer verfügbar. Dadurch bedingt ist allerdings ein gesondertes Handling des ersten Listen Elementes notwendig.

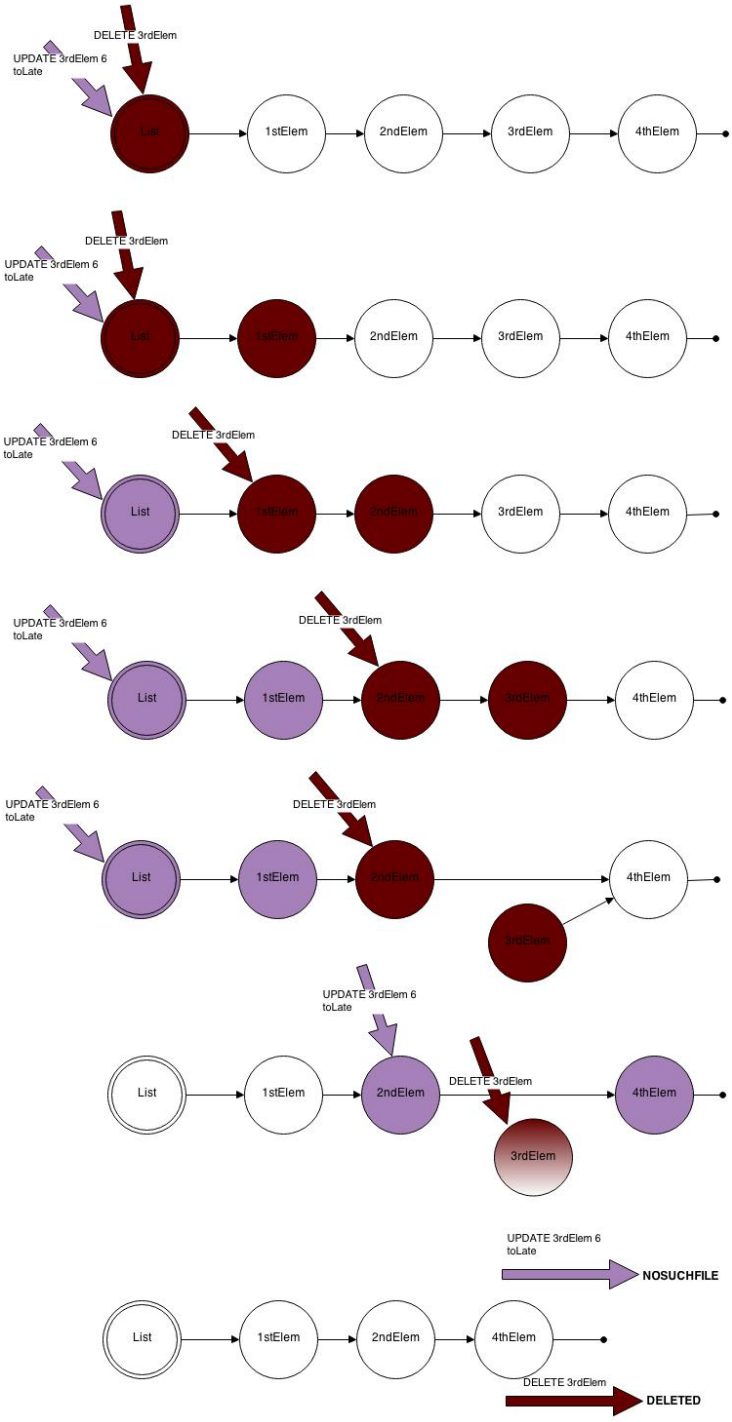

Abbildung 1: Ablauf gleichzeitiger Zugriffe

3.2.3 Input parsing

Ein nahezu ebenso grosses Problem, wie die geforderte Concurrency, stellte das Parsing der im Server auflaufenden Requests dar. Das vorgegebene Protokoll (A.1) beinhaltete nicht nur mehrere mögliche, in der Grösse dynamische Parameter, sondern auch mehrere mögliche Linien.

Eine Lösung mit viel *strlen(), strntok()* und *strncmp()* hätte ein nicht unerhebliches Mass an Komplexität und Fehleranfälligkeit mit sich gebracht. So ist die Funktion *strntok()* beispielsweise nicht Thread save. Daher wurde hierfür der Ansatz einer in Ragel geschriebenen State Machine gewählt.

Ragel bietet die Möglichkeit, mit wenigen Befehlen einen endlichen Automaten zu spezifizieren, welcher später in die Host-Language übersetzt wird. Dabei wird nicht nur eine wesentlich grössere Übersichtlichkeit des Parsings, sondern auch im Allgemeinen eine performantere Lösung als bei selbstständiger Implementation erzielt [5].

```
1  # Machine definition
2    list = 'LIST\n'  @list;
3    read = 'READ ' . filename . '\n' @read;
4    delete = 'DELETE ' . filename . '\n' @delete;
5    update = 'UPDATE ' . filename . ' ' . length . '\n' content . '\n
         ' @update;
6    create = 'CREATE ' . filename . ' ' . length . '\n' content . '\n
         ' @create;
7
8  main := (
9            list |
10           read |
11           update |
12           delete |
13           create
14         );
```

Listing 4: Input Parsing in *messageProcessing.rl*

Die obenstehenden Zeilen zeigen das Kernstück des Input Parsings. Das „main“-Statement bildet den Anfangszustand des Automaten, von welchem der Einstieg in in 5 andere endlichen Automaten möglich ist. Diese Automaten spezifizieren die möglichen Requests, wobei die Requests selbst wieder Automaten für „filename“, „length“, und „content“ enthalten, die während ihrer Abarbeitung alle Zeichen, die sie erhalten, für die spätere Verwendung in den Funktionen speichern. Sollte der Endzustand der Request behandelnden Automaten erreicht werden, wird die für die Verarbeitung des Inputs geeignete Funktion aufgerufen.

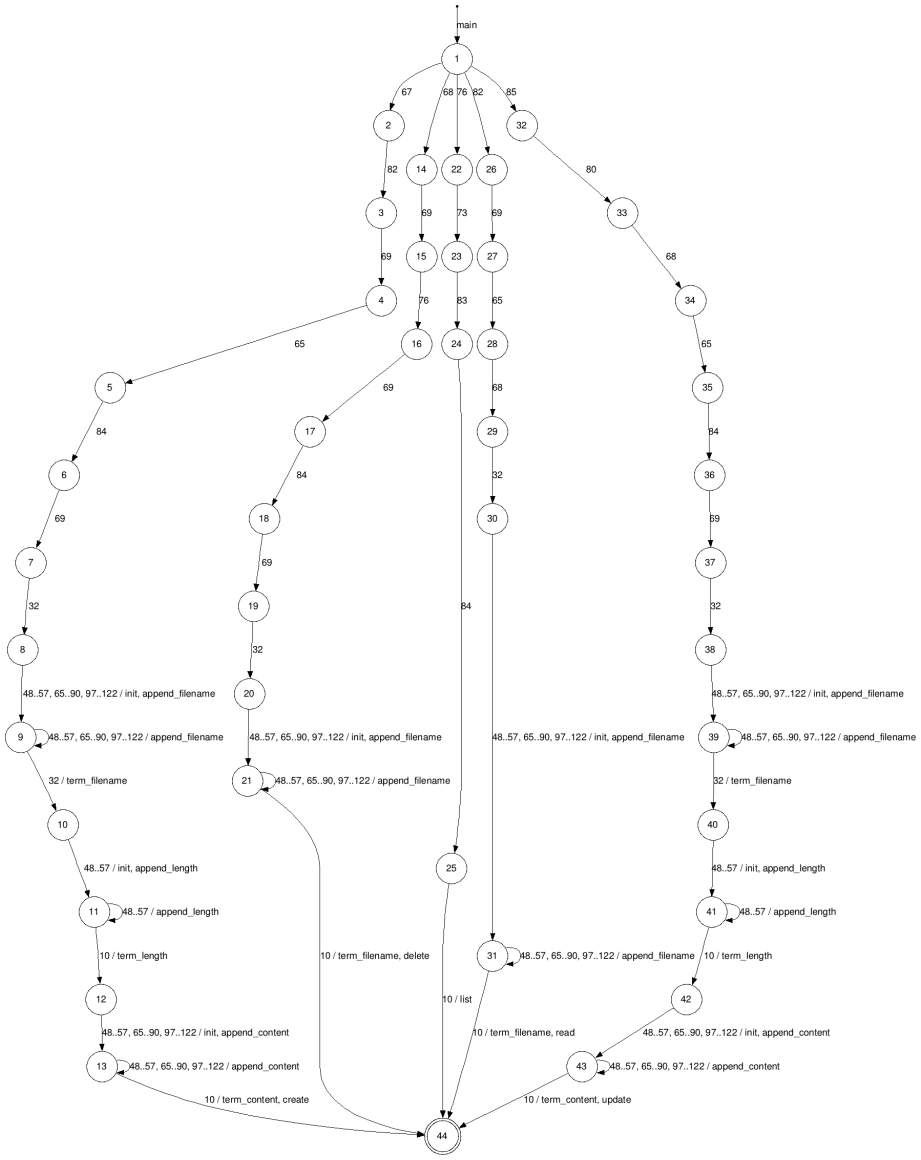

Abbildung 2: Darstellung der gesammten State Machine

Abbildung 2 ist eine detaillierte Übersicht aller möglichen Zustände während de Input Parsings bei der alle enthaltenen Automaten aufgelösst dargestellt wurden. Hierbei ist jedoch zu beachten, dass die Zahlen an den Zustandsübergängen als ASCII interpretiert werden müssen.

3.3 Test

Der automatisierte Modultest wurde ebenfalls in C implementiert. Er enthält unter Anderem:

- Test des Protokolls
  - Unbekannte Kommandos
  - Längen der Attribute
- Test der Funktionalitäten
  - Test aller angebotenen Funktionen
  - Test der Eindeutigkeit von Filenamen
- Concurrency Tests
  - Gleichzeitiges Erstellen u.Ä.
  - Nebenläufige Zugriffe

```
1 INFO: Testsuite done!
2 INFO: Ran 279 testcases
3 INFO: 279 tests succeded
4 INFO: 0 tests failed
5 INFO: Ran 5786 concurrent testcases
6 INFO: 5786 concurrent tests succeded
7 INFO: 0 concurrent tests failed
```

Listing 5: *Log eines erfolgreichen Tests*

3.4 Mögliche weitere Schritte

Durch die intensive Beschäftigung mit der Materie kamen im Laufe der Arbeit einige Punkte auf, deren Beachtung sich, wäre noch weitere Zeit vorhanden gewesen, angeboten hätte:

- Memory Leaks u.Ä.
  Nachdem das Konzept der Speicherverwaltung in C besser verstanden wurde, liegt die Vermutung nahe, dass einige Funktionen, so beispielsweise *join_with_seperator* aus *lib/termPaperLib.c,* mehr Speicher alloziieren als freigegeben wird. Ein Profiling mit Valgrind oder einem vergleichbaren Framework könnte diese Problem aufdecken.
- Performance
  Das Konzept neue Threads immer am Ende der Linked List anzufügen könnte bei sehr hohem Durchsatz unangenehm langsam werden. Die Betrachtung möglicher Optimierungen wäre hier sinnvoll.
- Code Style
  Gerade zu Anfang der Arbeit wurden viele Funktionsnamen usw. in CamelCase geschrieben, *const* nicht an allen möglichen Orten verwendet usw. eine Verschönerung des Codes wäre sehr sinnvoll. In diesem Rahmen wäre eine Beschäftigung mit Styles wie der Kernel Normal Form sehr spannend.

4 Fazit

Die Seminararbeit bot eine willkommene Möglichkeit sichtiefergehend mit einer grösseren Problemstellung in der Programmiersprache C zu beschäftigen. Obgleich C am Rande oft Erwähnung fand, war dies während des bisherigen Studiums noch nicht in einem solchen Rahmen möglich. Dadurch konnten verschiedene Konzepte, seien es richtige Makefiles, sinnvolle Strukturierung des Projektes mit Header-Files oder auch vieles Weiteres, in dieser Arbeit das erste Mal genauer betrachtet werden.

Die Anforderungen an Concurrency bereiteten dabei weniger Probleme, als das Handling der Programmiersprache an sich, da viele Konzepte der nebenläufigen oder gar verteilten Programmierung aus früheren Arbeiten - beispielsweise mit der Implementation des Map-Reduce Musters [2] - bekannt waren.

Die Verwendung von Ragel hingegen war absolut neu und sehr spannend. Da das Konzept sehr überzeugte, ist davon auszugehen, dass die Verwendung einer State Machine für die Implementation eines Protokolls im Rahmen dieser Arbeit, nicht die letzte war. Zumal Ragel auch in Kombination mit anderen Programmiersprachen, wie Ruby, Java oder Go, eingesetzt werden kann.

Neben zahlreichen Segmentation faults und Stunden des Pointer Debuggings, war die Bearbeitung der Aufgabe mit einer Menge von Aha-Erlebnissen verbunden. Viele abstrakt bekannte Konzepte aus anderen Vorlesungen konnten (endlich einmal) in der Praxis erlebt und nachvollzogen werden. So wird etwa bei der näheren Beschäftigung mit dem Memory Konzept in C, spätestens nach dem ersten „Stack smashing“, sehr eindrücklich der Unterschied von statisch alloziertem Stack und dynamisch alloziertem Heap bewusst.

Durch die Lektüre verschiedenster Quellen wurden des Weiteren auch nie explizit vermittelte Inhalte - zum Beispiel Elemente des für Linux C gängigen Code Style - erarbeitet und flossen in die Bearbeitung der Arbeit ein. Auch wenn einem erfahrenen C Programmierer wahrscheinlich an vielen Orten noch die Haare zu Berge stehen würden, war die Lernkurve im Rahmen der Arbeit, in der eigenen Wahrnehmung, enorm und führte hoffentlich zu einem guten und stabilem Ergebnis, bei dem Segmentation faults ausbleiben.

A Anhang

A.1 Protokoll [3]

```
List:
    Client sendet:
        LIST\n

    Server antwortet:
        ACK NUM_FILES\n
        FILENAME\n
        FILENAME\n
        FILENAME\n
        ...

    Server Beispiel:
        ACK 3
        abc
        def
        aei

Create:
    Client sendet:
        CREATE FILENAME LENGTH\n
        CONTENT

    Client Beispiel:
        CREATE abc 6\n
        Hello\n

    Server antwortet:
        FILEEXISTS\n
    oder
        FILECREATED\n

Read:
    Client sendet:
        READ FILENAME\n

    Client Beispiel:
        READ abc\n

    Server antwortet:
        NOSUCHFILE\n

    oder
        FILECONTENT FILENAME LENGTH\n
        CONTENT

    Server Beispiel:
        FILECONTENT abc 5\n
        1234\n
```

```
1  Update:
2      Client sendet:
3          UPDATE FILENAME LENGTH\n
4          CONTENT
5
6      Client Beispiel:
7          UPDATE abc 3\n
8          12\n
9
10     Server antwortet:
11
12         NOSUCHFILE\n
13
14     oder
15
16         UPDATED\n
17
18 Delete:
19     Client sendet:
20         DELETE FILENAME\n
21
22     Client Beispiel:
23         DELETE abc\n
24
25     Server antwortet:
26
27         NOSUCHFILE\n
28
29     oder
30
31         DELETED\n
```

Listing 6: Protokoll des Fileservers aus der Aufgabenstellung

A.2 Abbildungsverzeichnis

Abbildungsverzeichnis

1 Ablauf gleichzeitiger Zugriffe 9
2 Darstellung der gesammten State Machine 11

A.3 Listings

List of Listings

1 Held Text des Servers 4
2 Held Text des Modultests 5
3 Held Text des Clients 6
4 Input Parsing in *messageProcessing.rl* 10
5 *Log eines erfolgreichen Tests* 12
6 Protokoll des Fileservers aus der Aufgabenstellung 15

A.4 Literaturverzeichnis

Literatur

[1] Karl Brodowsky. bk1/sysprogramming-examples. `https://github.com/bk1/sysprogramming-examples`. (Visited on 05/18/2014).

[2] Reto Hablützel and Max Schrimpf. Github map-reduce. `https://github.com/mxmo0rhuhn/map-reduce`. (Visited on 06/16/2014).

[3] Nico Schottelius. Protokoll Fileserver. `https://raw.githubusercontent.com/telmich/zhaw_seminar_concurrent_c_programming/master/protokoll/fileserver`. (Visited on 05/18/2014).

[4] Dr. Adrian D. Thurston. Ragel Homepage. `http://www.complang.org/ragel/`. (Visited on 05/18/2014).

[5] Dr. Adrian D. Thurston. *Ragel State Machine Compiler*, 2013.