Seminararbeit

Fileserver in C

Seminar Concurrent Programming

vorgelegt beim Fachbereich Informatik

der Zürcher Hochschule für Angewandte Wissenschaften

in Zürich

von

Stefan Hauenstein

hauenste@studensts.zhaw.ch

Zürich

FS 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Seit Beginn der Systemprogrammierung und speziell seit der Einführung der Mehrprozessortechnologie ist das „Concurrent Programming“ ein wichtiger Aspekt geworden. Im Vordergrund steht aber nicht die Parallelisierung, sondern die Frage wie die vorhandenen Ressourcen gleichzeitig verwendet werden können ohne dass sie sich gegenseitig stören oder gar blockieren.

Die vorliegende Seminararbeit widmet sich diesem Thema.

1.1 Themenwahl

Zur Vereinfachung wurde durch den Dozenten zwei Themen vorgeschlagen. Zum einen einen Mehrbenutzereditor und zum anderen einen Dateiserver. Nach längerer Überlegung und Suchen nach möglichen Lösungsansätzen fiel die Wahl auf den Fileserver

1.2 Motivation

Die marginalen C-Kenntnisse zu Beginn der vorliegenden Arbeit vermochten keine richtige Motivation aufkommen. Mit der Zeit und mit der Vertiefung in das Thema des Concurrent Programming konnte man doch die ersten kleinen Erfolge sehen.

Auch kamen immer mehr Kenntnisse aus dem Fach Systemprogrammierung hinzu, welche sich in dieser Arbeit praktisch umsetzen liessen

1.3 Aufgabenstellung Fileserver

Allgemein müssen folgende Bedingungen erfüllt werden:

- kein globaler Lock
- Kommunikation via TCP/IP oder Unix Domain Socket
- fork + shm 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 Dozenten getestet

- Wenn die Eingabe valid ist, bekommt der Client ein OK
- Locking, gleichzeitiger Zugriff im Server lösen
- Client muss *nie* retry machen
- Alle Indices beginnen bei 0
- Debug-Ausgaben von Client/Server auf stderr

Des Weiteren gilt für den Server:

- Dateien sind nur im Speicher vorhanden
- Mehrere gleichzeitige Clients
- Lock auf Dateiebene
- Server muss nach CRUDL arbeiten (CREATE, READ, UPDATE, DELETE, LIST)

Hauptaufgabe beim Concurrent Programming für diesen Server ist die Harmonisierung der 5 CRUDL-Befehle. Sie dürfen sich nicht gegenseitig behindern und müssen auch für mehrere Clients gleichzeitig verfügbar sein ohne dass eine Sperrung über das ganze virtuelle Filesystem erfolgt.

1.4 Weiterführende Literatur

Nur mit dem vermittelten Unterrichtsstoff aus dem Fach Systemprogrammierung [1] wäre diese Arbeit nicht umsetzbar gewesen. Nur mit dem zusätzlichen Wissen aus den folgende Quellen war eine Umsetzung möglich: Dem Internet Portal „http://stackoverflow.com“ [2], mit seine hunderten von hilfreichen Fragen und Antworten rund um C Programmierung, den Büchern „Advanced Programming in the Unix Enviroment“ [3] und „C von A bis Z“ [4], wobei letzteres auch auf der Openbook Platform [5] erhältlich ist.

2 Lösungsansätze

Als erstes müssen Vorentscheidungen über die verwendeten Techniken getroffen werden. Der Socket-, Shared Memory/Semaphore- und der CRUDL Befehls-Teil bilden jedoch einzelne Segmente.

2.1 Lösungsansatz 1: Linked List mit Threads

2.1.1 PTHREAD und Mutex

Der Vorteil hier liegt im geteilten Adressraum aller Threads, was die Zuteilung erheblich vereinfacht. Auch ist seine Ausführung performanter. Mit MUTEX und „Conditions“ lassen sich die Zugriffe auf die kritischen Programmzeilen koordinieren. Sommit bilden PTHREAD und MUTEX eine unzertrennliche Einheit für das parallele Programmieren

2.1.2 Verkettete Liste

Das Prinzip der einfachverketteten Liste würde sich für dieses virtuelle Dateisystem sehr gut eignen. Die dynamische Anpassung und ein schnelles Suchverhalten, das bei einer doppelten verketteten Liste noch erhöht werden kann, sprechen für dieses Verfahren. Schwierigkeiten ergeben sich jedoch für die Befehle CREATE oder DELETE

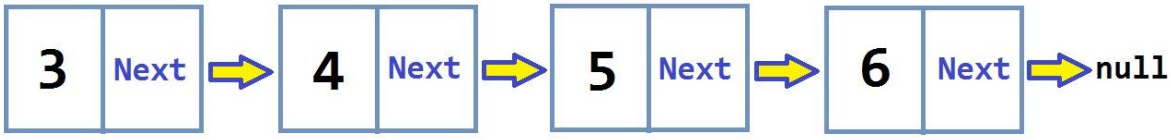

Abbildung 2.1: Einfachverkettete Liste

Schwierigkeiten ergeben sich jedoch für die Befehle CREATE oder DELETE. Wird gleichzeitig ein neuer Knoten angelegt und der benachbarte Knoten gelöscht, ist der neue Knoten zwar angelegt aber nicht in der Liste. Auch beim gleichzeitigen Anlegen zweier Knoten, geht einer verloren. Dies muss bei der Definition der Mutex-Architektur beachtet werden.

2.2 Lösungsansatz 2: Array mit FORK

2.2.1 FORK, Shared Memory und Semaphore

Der Kindprozess welcher durch das Forken erstellt wird, ist eine Kopie des Hauptprozesses. Er erhält seinen eigenen Adressraum, was ihn stabiler macht, aber auch unabhängig. Um trotzdem auf einen gemeinsamen Speicherbereich zuzugreifen zu können, wird ein „Shared Memory“ erstellt. In dieses Shared Memory lassen sich nun Variablen einbinden, welche für alle Kindprozesse und auch den Hauptprozess zugänglich sind. Koordiniert wird der Zugriff über Semaphore. FORK, Shared Memory und Semaphore bilden auch hier eine Einheit.

2.2.2 Array

Die Beschränkung der Grösse eines Arrays kann mit dem beschränkten Platz eines Speichermediums verglichen werden. Auch hier könnte eine einfachverkette Liste verwendet werden, das Array, da statisch, ist aber einfacher umsetzbar. Beachtet werden muss der DELETE Befehl und die daraus resultierende Verbindung zum UPDATE-Befehl. Da es sich um ein statisches Konstrukt handelt, können nicht dynamisch Teile aus der Mitte gelöscht werden. Diese müssen für einen Wiedergebrauch speziell Markiert werden.

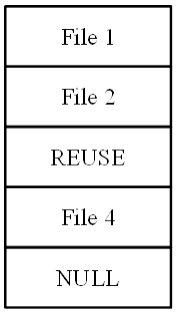

Abbildung 2.2: Array mit Markierung für Wiedergebrauch

2.3 Lösung für die Umsetzung

Die erste Wahl für die Umsetzung war Lösungsvariante 1. Eine dynamisch verkettete Liste und der gemeinsame Speicherraum liessen eine einfachere Implementierung vermuten. Das Grundgerüst war auch sehr schnell erstellt, leider konnte ich die Segmentierungsfehler während des Betriebs nicht beseitigen.

Dies veranlasste mich dazu zum zweiten Lösungsansatz zu wechseln, der trotz der komplexeren Semaphor-Einbindung letztendlich umsetzbar war.

3 Realisierung

3.1 Server

Der Server wurde auf die beiden Source-Dateien *server.c* und *shmsem.c*, sowie den dazugehörigen Header-Dateien *server.h* und *shmsem.h* aufgeteilt. Funktionen für die Erstellung und Initialisierung der Semaphore und des „Shared Memory“ sind in die Dateien *shmsem.h* und *shmsem.c* ausgelagert. Ursprünglich war auch die Auslagerung der CRUDL Befehle in eine eigene Header- und Source-Datei geplant, konnte aber nicht mehr umgesetzt werden. **Array (Dateien)**

Das Array soll ein Konstrukt von Dateien sein, wie es auf einer Festplatte zu finden ist, deshalb wurde zuerst eine Dateistruktur erstellt, in welche die benötigten Angaben für Namen, Grössen und Inhalte abgelegt werden können. Zusätzlich wurde noch eine Variable für die Semaphore angefügt (siehe Kapitel 3.3). Aus dieser Struktur wurde danach das Array erstellt.

3.3 SEMAPHORE

Wie in der Aufgabenstellung erwähnt, sollen lediglich „Dateien“ gelockt und kein globale Sperrung ausgelöst werden. Aus diesem Grund wurde in die Struktur der „Dateien“ die Semaphore Variable direkt eingefügt.

Zur Umsetzung einer Schreib- und Lesebedingung wurden Semaphore mit mehreren Slots verwendet. In der vorliegenden Konfiguration sind es 20 Slots. Jeder READ Befehl belegt einen Slot, bis alle belegt sind und gibt ihn nach Gebrauch wieder frei. Dagegen belegen die Befehle DELETE und UPDATE, welche in die „Datei“ schreiben, alle Slots auf einmal, damit kein weiterer Befehl Zugriff auf die „Datei“ erhalten kann. Nach dem Schreiben werden alle Slots wieder freigegeben.

3.4 CRUDL

Für jeden der fünf Befehle wurde ein „Use Case“ erstellt.

3.4.1 CREATE

Name	CREATE
Befehl Client	`CREATE FILENAME LENGTH\n CONTENT\n`
Akteure	• Client • Server
Beschreibung	File wird erstellt
Erfolgreicher Endzustand	File erstellt
Fehlgeschlagener Endzustand	File nicht erstellt
Vorbedingung	Keine
Nachbedingung	Keine
Ausgehende Nachricht bei Erfolg	FILECREATED\n
Ausgehende Nachricht bei existierendem File	FILEEXIST\n
Ablauf	1. Client Befehl senden 2. Server Befehl ausführen 3. Server Antwort senden
Fehlersituation	Befehl muss vollständig eingegeben sein.

3.4.2 DELETE

Name	DELETE
Befehl Client	`DELETE FILENAME\n`
Akteure	• Client • Server
Beschreibung	File wird gelöscht
Erfolgreicher Endzustand	File gelöscht
Fehlgeschlagener Endzustand	File nicht gelöscht
Vorbedingung	Keine
Nachbedingung	Keine
Ausgehende Nachricht bei Erfolg	`DELETED\n`
Ausgehende Nachricht bei Fehlschlag	`NOSUCHFILE\n`
Ablauf	1. Client Befehl senden 2. Server Befehl ausführen 3. Server Antwort senden
Fehlersituation	Befehl muss vollständig eingegeben sein.

3.4.3 READ

Name	READ
Befehl Client	`READ FILENAME\n`
Akteure	• Client

	• Server
Beschreibung	File wird gelesen
Erfolgreicher Endzustand	File gelesen
Fehlgeschlagener Endzustand	File nicht gelesen
Vorbedingung	Keine
Nachbedingung	Keine
Ausgehende Nachricht bei Erfolg	`FILECONTENT FILENAME LENGTH\n CONTENT`
Ausgehende Nachricht bei Fehlschlag	NOSUCHFILE\n
Ablauf	1. Client Befehl senden 2. Server Befehl ausführen 3. Server Antwort senden
Fehlersituation	Befehl muss vollständig eingegeben sein.

3.4.4 UPDATE

Name	UPDATE
Befehl Client	`UPDATE FILENAME LENGTH\n CONTENT\n`
Akteure	• Client • Server
Beschreibung	File wird geändert
Erfolgreicher Endzustand	File geändert
Fehlgeschlagener Endzustand	File nicht geändert
Vorbedingung	Keine
Nachbedingung	Keine
Ausgehende Nachricht bei Erfolg	`UPDATED\n`
Ausgehende Nachricht bei Fehlschlag	`NOSUCHFILE\n`
Ablauf	1. Client Befehl senden 2. Server Befehl ausführen 3. Server Antwort senden
Fehlersituation	Befehl muss vollständig eingegeben sein.

3.4.5 LIST

Name	LIST
Befehl Client	`LIST\n`
Akteure	• Client • Server

Beschreibung	Vorhandene Files werden aufgelistet
Erfolgreicher Endzustand	Liste
Fehlgeschlagener Endzustand	Keine Liste
Vorbedingung	Keine
Nachbedingung	Keine

Ausgehende Nachricht bei Erfolg	`ACK NUM_FILES\n`
	`FILENAME\n`
	`FILENAME\n`
	`FILENAME\n`
	`...`

Ausgehende Nachricht bei Fehlschlag	
Ablauf	1. Client Befehl senden 2. Server Befehl ausführen 3. Server Antwort senden
Fehlersituation	Befehl muss vollständig eingegeben sein.

3.5 Aufgetretene Probleme

Hauptsächliches Problem bei der Erstellung des Programms, waren die immer wieder auftretenden Segmentierungsfehler, die aber hauptsächlich durch das falsche Verwenden von Pointern oder Pointer von Pointern entstanden. Dies konnte aber durch die Änderung des Lösungsansatzes auf ein Array mit fester Grösse behoben werden.

Ein weiteres Problem war die Implementierung der Semaphore. Was bei einem Kindprozess noch als lauffähig aussah, konnte leider mit mehreren Kindprozessen nicht mehr umgesetzt werden.. Grund war eine Auftretende Diskrepanz zwischen mehreren gleichen Befehlen auf dieselbe „Datei“. Erst durch eine weitere Aktualisierung der Semaphore durch *semctl()* vor der Rücksetzung vermochte dieses Problem zu lösen.

4 Fazit

Die grösste Schwierigkeit dieser Seminararbeit stellte das fehlende Wissen in der Programmiersprache C dar. Bevor auch nur eine Zeile Code geschrieben werden konnte, musste einiges an zusätzlichem Wissen angeeignet werden. Das rudimentäre Wissen aus den Einführungswochen reichte bei weitem nicht aus, um ein lauffähiges Programm zu erstellen. Auch das Fach Systemprogrammieren konnte nur langsam Licht ins Dunkle bringen.

Der erste Ansatz mit Threads und einer verlinkten Liste war ein sehr schöner Ansatz, scheiterte aber an den nicht zu behebenden Fehler, die während des Betriebs auftraten. Ohne hinreichende Kenntnisse im Umgang mit einer „Debugging Software“ erschien ein Wechsel zum zweiten Lösungsansatz angebracht

Die intensive Auseinandersetzung mit Der Materie steigerte zweifellos die Motivation und der Ehrgeiz das Programm fertigzustellen. Der Enorme Zeitaufwand, der dadurch entstand, und weit über den veranschlagten 60 Stunden lag, steht jedoch in keinem Verhältnis zum Ergebnis.

Literaturverzeichnis

[1] K. Brodowsky, Interviewee, *Systemprogrammierung.* [Vorlesung]. Februar bis Juni 2014.

[2] „stackoverflow.com“ Stack Exchange, Inc., 2014. [Online]. http://stackoverflow.com. [Mehrere Zugriffe im FS 2014].

[3] W. R. Stevens und S. A. Rago, Advanced Programming in the Unix Environment 3th Edition, Boston: Addison-Wessley, 2013.

[4] J. Wolf, C von A bis Z: Das umfassende Handbuch, Bonn: Galileo Computing, 2009.

[5] G. Comouting, „OpenBooks“ Galileo Computing, 23 März 2014. [Online]. http://openbook.galileocomputing.de. [Mehrere Zugriffe im FS 2014].