Fileserver

	ZHAW Zürich
Seminar:	**Concurrent C**
Dozent:	**Nico Schottelius**
Autor:	**Miroslav Mirkovic**
Datum:	**22.06.2014**

Änderungsnachweis

Datum:	Version:	Vorgenommene Änderung:	Autor:
27.05.2014	0.0.1	Dokumentenstruktur festgelegt.	M.M.
28.05.2014	0.1.0	Einleitung, Server Protokoll und Server Entwurf	M.M.
21.06.2014	0.6.0	Anleitung, Umsetzung und Fazit	M.M.
21.06.2014	0.9.4	Anhänge Formuliert und Soll-Zeitplanung eingefügt.	M.M.
21.06.2014	0.9.9	Quellenangaben und Glossar fertiggestellt.	M.M.
22.06.2014	1.0.0	Review, Zeitplanung und Korrekturlesung	M.M.

Inhaltsverzeichnis

Einleitung

Ausgangslage

Im Rahmen des Concurrent C Seminars, hatten wir die Möglichkeit einen Fileserver oder Multi-User-Editor zu erstellen. Ich habe mich für die Fileserver Variante entschieden.

Ziel der Arbeit

Die Methoden des Concurrent-Programming in C kennenlernen und einen lauffähigen Server, welcher in der Lage ist Dateien im Arbeitsspeicher zu verwalten sowie konkurrierenden Zugriff auf diese zu ermöglichen.

Aufgabenstellung

1. Dateien sind nur im Speicher vorhanden (Echtes Dateisystem darf nicht verwendet werden)

2. Mehrere gleichzeitige Clients

3. Lock auf Dateiebene (kein globaler Lock)

4. Kommunikation via TCP/IP

5. Fork und Shared Memory (jede Verbindung startet eigenen Prozess / Hauptprozess kann bind-list-accept machen)

6. Server sendet OK an client sofern die Eingabe valide ist (Locking wird Serverseitig gelöst, Client muss nie retry machen)

7. Indices beginnen bei 0

8. Debug-Ausgaben von Client/Server auf Stderr

Erwartete Resultate

- Lauffähiger Server, welcher durch ein Testskript vom Dozenten getestet werden kann. (in Git Repo: https://github.com/MiroslavM/concurrent_c)

- Make Datei, welche run erstellt (Serverbinary)

- Dokumentation der Arbeit (doc.pdf)

- Präsentation (10 - 15min)

Vordefiniertes Server-Protokoll

List:	Client:	„LIST\n“
	Server:	„ACK #FILES\n“ „FILENAMEXX\n“ …
	Beispiel:	„ACK 3“ „abc“ „cde“ „efg“
Create:	Client:	„CREATE FILENAME LENGTH\n“ „CONTENT“
	Server:	„FILE EXISTS\n“ || „FILE CREATED\n“
	Beispiel:	„CREATE abc 6\n“ „Hello\n“
Read:	Client:	„READ FILENAME\n“
	Server:	„NOSUCHFILE\n“ || „FILECONTENT FILENAME LENGTH\n“ „CONTENT“
	Beispiel:	„READ abc\n“ -> „FILECONTENT abc 6\n“ „Hello\n“
Update:	Client:	„UPDATE FILENAME LENGTH\n“ „CONTENT“
	Server:	„NOSUCHFILE\n“ || „UPDATED\n“
	Beispiel:	„UPDATE abc 3\n“ „12“
Delete:	Client:	„DELETE FILENAME\n“
	Server:	„NOSUCHFILE\n“ || „DELETED\n“
	Beispiel:	„DELETE abc\n“

Weitere Projekt-Anforderungen

- Projekt ist auf github zu finden (https://github.com/MiroslavM/concurrent_c)
- Applikation lauffähig unter Linux
- Nach Ausführung von „make“ existieren:
  „run“: Binary des Servers (soll nicht abstürzen / keine SEGV auftreten)
  „test“: Testprogramm zum prüfen der definiert Funktionalität siehe Protokoll
  „doc“: PDF Dokumentation
- kein globaler Lock
- Kommunikation via TCP/IP oder Unix Domain Sockets
- fork und shared-memory (für jede Verbindung ein prozess/thread)
- Hauptthread kann bind/listen/accept machen
- Fokus liegt auf dem Serverteil
  Client ist zum Testen da, Server wird durch Skript vom Dozenten getestet.
- Wenn die Eingabe valid ist, bekommt der Client ein OK
  Locking, gleichzeitigen Zugriff im Server lösen
  Client muss nie retry machen
- Alle Indices beginnen bei 0
- Debug-Ausgaben von Client/Server auf stderr
- Dateien sind nur im Speicher (Shared-Memory) vorhanden FS nicht benutzen.
- Mehrere gleichzeitige Clients
- Lock auf Dateiebene

Anleitung zur Nutzung

Der Server wird mit der Eingabe von „./run“ gestartet. Beendet wir der Serverprozess mit CTRL+C (Signal). Eventuelle Fehlermeldungen werden, wie in den Anforderungen gefordert auf stderr ausgegeben.

Das Testen des Servers wird mit dem „Test“ Programm durchgeführt. Gestartet wird der Prozess mit Eingabe von „./test BEFEHL Parameter“. Folgende Befehle sind vorhanden:

- LIST
- CREATE
- READ
- UPDATE
- DELETE

Der Parameter ist die Pfadangabe zu einer vorhandenen Textdatei. Dieser Parameter wird besonders überprüft, zum einen, ob die Datei lesbar ist zum anderen, ob die Datei tatsächlich eine Textdatei ist.

Falls der Benutzer das Programm fehlerhaft bedient, wird ihm die Beschreibung zur Bedienung des Programms angezeigt, dies wird ebenfalls angezeigt falls ein benötigter Parameter fehlt.

Server-Entwurf

Kommunikation

Für die Kommunikation zwischen Server und Clients wird das TCP Protokoll verwendet. Hierbei wartet der Server auf eingehende Verbindungen, falls eine Verbindung vorliegt, wird diese an Child Prozess weitergegeben und der Server geht zurück in den Zustand wartend. Dies soll Verhindern, dass ein Client warten muss.

Organisation der Dateien

Die Dateien werden in Shared-Memory Segmenten gespeichert. Die Informationen zu den Dateien werden im FileInfo Struct festgehalten. Die Structs sind folgendermassen aufgebaut:

Struct FileInfo { fileName, fileSize, shmid, sem, *prevFile, *nextFile }

Die Structs sollen in dynamisch (doppelt) verketteten Listen festgehalten werden und Alphabetisch nach dem Dateinamen (fileName) sortiert werden, dies soll die Last und Zeit auf ein Minimum reduzieren, wenn eine Datei gesucht/bearbeitet werden muss. Ausserdem sollte dieser Lösungsansatz, die dynamische Allozierung von Shared Memory Segmenten für neue oder zu verändernde Dateien vereinfachen.

Definition von Server-Funktionen

void listFiles()
Alle vorhandenen Dateien werden aus dem SharedMemory ausgelesen und die Anzahl, sowie die Namen an den Client zurückgegeben.

void createFile(filename, filesize, content)
Diese Funktion kümmert sich darum, die Datei in das SharedMemory zu laden.

void readFile(filename)
Wenn der Client read aufruft, wird der Inhalt der Datei aus dem SharedMemory gelesen und an den Client gesendet.

void updateFile(filename, filesize, content)
Bestehende Datei wird aus dem SharedMemory gelöscht und die neue Datei in das gleiche Segment geschrieben.

void deleteFile(filename)
Das entsprechende SharedMemory Segment wird gelöscht und die Struktur entfernt.

void stop()
Wenn das Signal zum Beenden des Programms empfangen wird, werden alle Ressourcen freigegeben, erst danach wird das Programm beendet.

Umsetzung

Daemon

Der Serverprozess läuft als Daemon, die Umsetzung des Daemons selbst war einfach zu bewerkstelligen aber die Kontrolle des Daemonprozesses gestaltete sich ein wenig komplizierter. Der Prozess schreibt die Prozess ID in die run.pid Datei, welche sich im gleichen Ordner befindet wie das Programm selbst, um Festzustellen ob der Prozess bereits läuft, wird nach dieser Datei gesucht, falls diese vorhanden ist, wird die enthaltene PID als Parameter für die Funktion kill(pid,0) genommen, welche ESCRH zurückgibt, falls der Prozess nicht läuft in diesem Fall wird ein Prozess gestartet. Vom Einsatz des Daemons habe ich abgesehen, da gefordert ist, die Fehlermeldungen auf den Terminal auszugeben, dies macht bei einem Daemon aber wenig Sinn.

TCP Socket

Bei Server und Clientprozess werden jeweils Sockets erstellt, Die Verbindung basiert auf IPv4 sowie TCP (welches gewährleistet, dass die abgeschickten Daten auch ankommen). Befehle und Resultate werden in einer Zeile bidirektional von Server zu Client gesendet.

Shared Memory

Die Empfangenen Daten werden, nach entsprechender Bearbeitung in den dazugehörigen Funktionen, in Shared Memory Segmenten gespeichert. Der Zugriff auf die Segmente wird durch Semaphore geregelt. Dadurch wird verhindert, dass durch konkurrierende Zugriffe, die gleichen Segmente von mehreren Kindprozessen bearbeitet und somit zu inkonsistenten Zuständen gebracht werden.

Dynamisch verkettete Listen

Die FileInfo Structs werden in einer doppelt verketteten Liste verwaltet. Die Liste wird alphabetisch sortiert (nach dem entsprechenden FileName member des FileInfo Structs) damit das Suchen nach dem entsprechenden FileInfo Ressourcenschonend durchgeführt werden kann. Bei der Verwaltung der verketteten Liste kommen ebenfalls Semaphore zum Einsatz (Hinzufügen und Entfernen von Structs wird so vor konkurrierenden Zugriffen geschützt). Aus Zeitgründen wurde darauf verzichtet die Liste sortiert anzulegen, kann aber als Erweiterung des Programms später implementiert werden.

Fazit

Dies war das erste Projekt, welches ich in der Programmiersprache C umsetzen durfte (eines der wenigen Programmierprojekte überhaupt). Sie gab mir die Gelegenheit, das eher theoretisch gefasste Wissen, aus dem Kurs zur Systemprogrammierung, anzuwenden. Die Arbeit hat vor allem dann Spass gemacht, wenn die Probleme einfach gelöst werden konnten, manchmal zog sich aber die Umsetzung von Programmieraufgaben unnötig hin, was wenig zur Motivation beigetragen hat. Vor allem die Aufbereitung der empfangenen Daten, hat mir Probleme bereitet. Auch die Umsetzung von Semaphoren, gestaltete sich schwieriger als erwartet. Die Zeitvorgabe von 60h erlaubte mir (mit meinen bescheidenen Programmierfähigkeiten) nicht das Programm optimal zu gestalten, aber die vorgegebenen Funktionen bietet es an. Ich wollte gerne die dynamisch verkettete Liste sortiert verwalten, musste aber von der Implementierung absehen. Auch habe ich einige Zeit dafür aufgewendet den Server als Daemon laufen (und stoppen) zu lassen und erst spät feststellte, dass dies nicht der optimale Lösungsweg war, da die Fehlermeldungen des Server auf Stderr ausgegeben werden sollten.

Dies war auf jeden Fall eine sehr wertvolle Erfahrung und eine fordernde Aufgabe. Der Lerneffekt war sehr hoch, vor allem weitere Möglichkeiten zu evaluieren das Programm optimaler zu gestalten, hat mir einige aufschlussreiche Momente beschert.

Anhang

Glossar

Begriff	Erklärung
Concurrent	Konkurrierend (Mehrere Prozesse arbeiten die Anfragen der Clients ab)
File Server	Dateiserver (Dateien werden in Shared Memory Segmenten gespeichert)
TCP IP	Netzwerk Protokollfamilie (Kümmern sich um Dateiübertragung im Netzwerk).
Shared Memory	Speicher, der von unterschiedlichen Prozessen angesprochen werden kann. Siehe: http://de.wikipedia.org/wiki/Shared_Memory
Locking	Verhindern von gleichzeitigem Zugriff auf gemeinsames Betriebsmittel. Bsp. Shared Memory
Stderr	Datenstrom im Unix und ähnlichen Betriebssystemen. Siehe auch Stdin / Stdout
SEGV	Segmentation Violation – Zugriffsverletzung (Zugriff auf nicht vorhandenes Segment im Shared Memory)
Signale	Werden in Unix ähnlichen Systemen für die Signalisierung von Fehlern an Prozesse benötigt.
Structs	Sind komplexe Datentypen in der Programmiersprache C, welche sogenannte Member (also verschiedene Variablentypen) beinhalten / zusammenfassen können.
Socket	„Stecker“ der für die IPC verwendet wird (Intern und im Netzwerk)
IPC	Inter Process Communication (Austausch von Daten zwischen Prozessen oder Threads)

Literaturverzeichnis / Links

Advanced Programming in the UNIX Environment [W. Richard Stevens, Stephen A. Rago]

Informationen zur Entwicklungsumgebung:
www.github.com
http://www.geany.org/

Informationen zu ‘Make’:
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
http://www.gnu.org/software/make/manual/make.html
http://de.wikipedia.org/wiki/Makefile

Informationen zu C Programmierung:

GIT Repository:
https://github.com/telmich/zhaw_seminar_concurrent_c_programming

Dynamische Liste von Structs (um die Clientinformationen zu verwalten):
http://openbook.galileocomputing.de/c_von_a_bis_z/021_c_dyn_datenstrukturen_002.htm#mj0230cca6aa7fb87181f7f69e396d6fd8
http://perlgeek.de/de/artikel/einfach-verkettete-listen

Struct definition
http://de.wikibooks.org/wiki/C-Programmierung:_Komplexe_Datentypen
LinkedList
http://www.cprogramming.com/tutorial/c/lesson15.html
Sorting LinkedList
https://answers.yahoo.com/question/index?qid=20120405183650AATOzJ4

Daemon
http://stackoverflow.com/questions/17954432/creating-a-daemon-in-linux
http://www.thegeekstuff.com/2012/02/c-daemon-process/

TCP Socket Verbindung
http://openbook.galileocomputing.de/c_von_a_bis_z/025_c_netzwerkprogrammierung_006.htm
http://books.google.ch/books?id=dPk2MMUq5B8C&pg=PA107&lpg=PA107&dq=int+server_socket&source=bl&ots=2mHdCIfPIe&sig=f8TAlHLt2P0hNcD1y13V1SGzths&hl=de&sa=X&ei=doihU-HxJciY0QXYxYDIDw&ved=0CHgQ6AEwCQ#v=onepage&q=int%20server_socket&f=false
http://www.tutorialspoint.com/unix_sockets/socket_server_example.htm
http://shoe.bocks.com/net/#send

Verkettete Listen
http://openbook.galileocomputing.de/c_von_a_bis_z/021_c_dyn_datenstrukturen_002.htm#mj0230cca6aa7fb87181f7f69e396d6fd8
http://perlgeek.de/de/artikel/einfach-verkettete-listen

Manipulation am Shared Memory
http://openbook.galileocomputing.de/unix_guru/node393.html
http://openbook.galileocomputing.de/linux_unix_programmierung/Kap09-006.htm
http://www.cs.cf.ac.uk/Dave/C/node27.html

Zeitplanung

Systemprogrammierung

Company: ZHAW
Manager: Miroslav Mirkovic
Start: June 15, 2014
Finish: June 20, 2014
Report Date: June 21, 2014

Gantt Chart

WBS	Name	Work	Week 25, 2014							Week 26, 2014							Week 27, 2014							Week 28, 2014						
			15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	**Dokumentation**	**1d**																														
1.1	Einleitung																															
1.2	Server Protokoll und Entwurf																															
1.3	Anleitung Umsetzung Fazit	3h																														
1.4	Anhänge Zeitplanung Quellenangaben	2h																														
1.5	Review Korrekturlesung	1h																														
2	**File Server**	**7d**																														
2.1	Kommunikation	4h																														
2.2	Parser und Funktionsaufrufe	1d 4h																														
2.3	Shared Memory und Dateiverwaltung	3d																														
2.4	Verlinkte Liste	2d																														
3	**Test Programm**	**1d 7h**																														
3.1	Parser für Befehle	3h																														
3.2	Kommunikation	4h																														
3.3	Dateien auslesen und senden	1d																														

Tasks

WBS	Name	Start	Finish	Work	Complete
1	**Dokumentation**	**Jun 20**	**Jun 20**	**1d**	
1.1	Einleitung	Jun 20	Jun 20		0%
1.2	Server Protokoll und Entwurf	Jun 20	Jun 20		100%
1.3	Anleitung Umsetzung Fazit	Jun 20	Jun 20	3h	100%
1.4	Anhänge Zeitplanung Quellenangaben	Jun 20	Jun 20	2h	100%
1.5	Review Korrekturlesung	Jun 20	Jun 20	1h	100%
2	**File Server**	**Jun 16**	**Jun 18**	**7d**	
2.1	Kommunikation	Jun 16	Jun 16	4h	100%
2.2	Parser und Funktionsaufrufe	Jun 16	Jun 17	1d 4h	100%
2.3	Shared Memory und Dateiverwaltung	Jun 16	Jun 18	3d	100%
2.4	Verlinkte Liste	Jun 16	Jun 17	2d	100%
3	**Test Programm**	**Jun 16**	**Jun 16**	**1d 7h**	
3.1	Parser für Befehle	Jun 16	Jun 16	3h	100%
3.2	Kommunikation	Jun 16	Jun 16	4h	100%
3.3	Dateien auslesen und senden	Jun 16	Jun 16	1d	100%