Zürcher Hochschule für Angewandte Wissenschaften

Seminar Concurrent Programming

# File-Server in C anhand CRUDL

*Author:*
Philipp Schalcher

*Betreuer:*
Nicolas Schottelius

27. Mai 2014

# Danksagung

# Inhaltsverzeichnis

# Zusammenfassung

# Einleitung

Ein wichtiger Teil der Systemprogrammierung ist das Concurrent Programming. Dabei steht nicht die Parallelisierung im Vordergrund, sondern das Problem, von Zugriffen auf Ressourcen und die damit auftretenden Schwierigkeiten. Gleichzeitige Bearbeitungen von Ressourcen können, je nach ausgeführter Arbeit, zu schwerwiegenden Problemen führen. Wird zum Beispiel eine Datei gelöscht, obwohl die Datei momentan an einem anderen Ort geöffnet ist und das System dies nicht bemerkt, tritt ein Konflikt auf. Soll die Datei nun gelöscht werden und der Bearbeiter verliert den Zugriff, oder soll die Datei einfach nicht gelöscht werden, mit dem Vermerk, dass die Datei geöffnet ist?
In dieser Arbeit soll nun ein Dateiserver in C programmiert werden, der genau solche Probleme abfangen soll. Dabei gilt, dass der Server nach dem Schema CRUDL funktionieren soll. CRUDL steht für:

- CREATE
- READ
- UPDATE
- DELETE
- LIST

Je nach aufgerufenem Task, darf eine Datei nicht mehr zugreifbar sein.

Die Arbeit soll keine realen Dateien verarbeiten können. Das Ziel liegt in der Realisierung des CRUDL-Systems. Die Dateien selber werden in einem Shared-Memory gespeichert (nur die Dateinamen). Sobald der Server gestoppt wird, gehen die Dateien im Shared-Memory verloren.

# Kapitel 1

# Anleitung

Der Server kann über die Binary “run”gestartet werden. Im Konsolenfenster werden Nachrichten zur Initialisierung ausgegeben. Sobald der Server gestartet ist, kann über das Binary “test”ein Testclient gestartet werden. Über diesen können Befehle an den Server gesendet werden.

## 1.1 Befehle

Der Server erkennt folgende Befehle:

- CREATE Dateiname Grösse
- READ Dateiname
- UPDATE Dateiname Grösse
- DELETE Dateiname
- LIST

## 1.2 CREATE

Syntax: CREATE *Dateiname Grösse*
CREATE erstellt eine neue Datei im Fileserver. Als Parameter wird der gewünschte Dateiname und die Grösse als Zahl mitgegeben. Falls die Datei schon vorhanden ist, gibt der Server ein “FILEEXISTS”zurück. Andernfalls wird bei Erfolg die Nachricht “FILECREATED”an den Client gesendet.

Beispiel: CREATE meineErsteDatei 30

## 1.3 READ

Syntax: READ *Dateiname* READ liest den Inhalt einer Datei aus und sendet sie an den Client. Falls der Dateiname nicht existiert, wird die Nachricht "NOSUCHFILE"zurückgegeben.

Beispiel: READ meineErsteDatei

## 1.4 UPDATE

Syntax: UPDATE *Dateiname Grösse*
Mit UPDATE wird die vorhandene Datei geändert. Dabei wird die Grösse und der Inhalt verändert. Existiert der angegebene Dateiname nicht, wird die Nachricht "NOSUCHFILE"zurückgegeben. Ist der Befehl erfolgreich wird in einem zweiten Schritt der Inhalt verlangt.

Beispiel: UPDATE meineErsteDatei 13

## 1.5 DELETE

Syntax: DELETE *Dateiname*
Nach dem Aufruf von DELETE wird die ausgewählte Datei gelöscht. Dabei gibt der Befehl die Meldung "NOSUCHFILE"zurück, falls kein Eintrag mit dem Dateinamen bekannt ist. Ansonsten erhält man die Nachricht "DELETED".

Beispiel: DELETE meineErsteDatei

## 1.6 LIST

Syntax: LIST LIST veranlasst den Server dazu, alle Dateien zu zählen und deren Namen an den Client zu senden.

# Ausgangslage

Auf Wunsch des Dozenten in Concurrent Programming soll entweder ein Datei-Server oder ein Mehrbenutzereditor in C programmiert werden. Diese Arbeit behandelt die Aufgabe mit dem Datei-Server. Dabei muss der Server gewisse Bedingungen erfüllen. Diese wären:

- Keine SEGV während des Betriebs.
- Jeder Client der verbindet, soll mit einem eigenen Prozess behandelt werden.
- Server muss nach CRUDL arbeiten.
- Das echte Dateisystem darf nicht genutzt werden.
- Dateien sind nur im Speicher vorhanden.

CRUDL bedeutet soviel wie CREATE, READ, UPDATE, DELETE und LIST. Diese 5 Befehle müssen vom Server umgesetzt werden. Der Fokus liegt dabei auf den Zugriff von verschiedenen Prozessen auf Dateien (als Ressourcen). Gewisse Operationen wie DELETE müssen Dateien komplett sperren, damit diese während der Operationen nicht von anderen Operationen besetzt werden können. Das heisst, während eines DELETE darf kein READ, UPDATE oder LIST durchgeführt werden. Innerhalb dieser Befehle gibt es mehrere Kombinationen solcher Sperrungen. Diese müssen abgefangen werden.

## Lösungsansatz

Die Problematik besteht für mich aus 3 Teilproblemen. Das erste Problem ist die Netzwerkverbindung und die darauf folgende Erstellung eines neuen Prozesses, der die Anfragen annimmt und verarbeitet. Als Lösung würde ich daher auf eine normale TCP/IP Verbindung setzen. Diese wird über Sockets gelöst. Sobald ein Client verbindet, wird per fork() ein neuer Prozess erstellt, der als Child-Prozess die Anfragen des Clients annimmt und bearbeitet. Innerhalb des Child-Prozesses müssen daher die Befehle abgefangen werden und die eigentlichen Aktionen ausgeführt werden.
Als zweites Problem sehe ich die Speicherung der "Dateien". Hier würde ich so vorgehen, indem ich eine Struktur aufbaue, die für jede Datei ihren Dateinamen, ihre Grösse und den Inhalt speichert. eine einzelne Struktur soll also eine Datei symbolisieren. Um mehrere Dateien speichern zu können, wird aus der Struktur ein Array erstellt. Allerdings begrenzt

ein Array die Anzahl der zu verarbeitenden Dateien. Allerdings könnte dies gleichgesetzt werden mit dem Plattenspeicher, der irgendwann auch voll ist.
Als drittes Problem ist die Sperrung der Dateien ein zentraler Punkt. Da für den Server keine globalen Sperren genutzt werden dürfen, würde ich Semaphore einsetzen, die die einzelnen Codeteile sperren und so von Prozessen nicht gleichzeitig ausgeführt werden können. Der Semaphor an sich muss aber mehr als ein "Ticket"vergeben können, da zum Beispiel READ mehrfach ausgeführt werden darf. Allerdings muss ein DELETE den kompletten Semaphor besetzen, damit keine anderen Operationen durchgeführt werden können. Auch hier gilt wieder, das mit der Semaphorengrösse auch die Anzahl Clients beschränkt wird. So können bei einem Semaphor mit 10 Tickets maximal 10 Clients gleichzeitig einen READ durchführen. Auch wenn 20 Clients verbunden wären, so würden doch immer nur 10 bedient werden.

# Realisierung

## 1.7 Netzwerk

### 1.7.1 Testing

# Schwierigkeiten

# Fazit