Zürcher Hochschule für Angewandte Wissenschaften

Seminar Concurrent Programming

File-Server in C anhand CRUDL

Author:
Philipp Schalcher

Betreuer:
Nicolas Schottelius

30. 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.

1.7 Beschränkung

Durch den Einsatz der Semaphore ist der Server momentan auf maximal 10 gleichzeitige Read-Vorgänge eingeschränkt. Dies könnte mit grösseren Semaphor verändert werden. Da aber keine Angaben im Auftrag gemacht wurden, werde ich die Einstellung in diesem Zustand belassen.

Kapitel 2

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.

2.1 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.

Kapitel 3

Realisierung

In diesem Kapitel werde ich auf die Realisierung eingehen. Ein kurzer Abschnitt handelt von meiner Vorgehensweise und wie ich den Server aufgebaut habe. Schwerpunkt wird allerdings auf den Bereich "Schwierigkeiten" und "Testing" gelegt.

3.1 Implementation

Der ganze Server ist in einem C-Sourcefile zu finden. Der Aufbau ist relativ einfach:

- Includes der Bibliotheken
- Globale Variablen definieren
- Funktionen definieren
- Main-Funktion definition
  - Erstellung des Signal-Handler
  - Erstellung des Shared-Memory
  - Erstellung des Semaphore
  - Aufbau des Netzwerkteils
  - Abfangen von Verbindungsanfragen und Prozesserstellung
    - Abfangen des LIST-Befehls
    - Abfangen des CREATE-Befehls
    - Abfangen des READ-Befehls
    - Abfangen des UPDATE-Befehls
    - Abfangen des DELETE-Befehls

Die komplette Logik ist in den Bereichen der Befehlsabfragen zu finden. Die Dateien werden in einer definierten Struktur gespeichert, welche als Array auf das Shared-Memory gelegt wird. Sobald ein freier Platz oder eine bestimmte Datei gesucht werden muss, durchläuft das Array eine Schlaufe bis die Länge des String im Namen gleich 0 ist. So wissen wir, dass wir die letzte Position erreicht haben. Wie die Befehle genau implementiert sind, wird in den folgenden Abschnitten erklärt.

3.1.1 Implementation LIST

Für den Befehl List, wird das Array durchgelaufen und die Anzahl gefundenen Dateien gezählt. Da auch Einträge mit Namen "EMPTY"vorkommen können (mehr dazu im Kapitel "Implementation DELETE"), werden diese ignoriert. Die korrekten Dateinamen werden an einen vorhanden String angehängt. Ist die letzte Position im Array erreicht, wird die Nachricht generiert und an den Client gesendet. Danach steht der Server wieder für die nächste Eingabe zur Verfügung.

3.1.2 Implementation CREATE

Das Array wird wieder durchgegangen bis das Ende erreicht ist (wieder über die Längenprüfung von Name) oder ein Eintrag mit "EMPTY"gefunden wird. Ist der vorgegebene Dateiname schon vorhanden, bricht der Vorgang ab. Ansonsten wird der Name, die Grösse, der Semaphorwert gesetzt und der Client wird nach dem Inhalt gefragt. Der Client muss den Inhalt eingeben, welcher dann im Feld Content gespeichert wird.

3.1.3 Implementation READ

Das Array wird nach dem angegebenen Namen durchsucht. Wird kein Eintrag gefunden, erhält man ein "NOSUCHFILE". Ansonsten holt sich der Prozess ein Semaphorticket und liest den Inhalt. Der Inhalt wird dann an den Client gesendet. Zum Abschluss gibt der Prozess das Semaphorticket wieder zurück. Sind alle Tickets verbraucht, wartet der Prozess bis eines frei wird.

3.1.4 Implementation UPDATE

Der erste Teil ist gleich wie im READ-Bereich. Ist das spezifische File gefunden, werden zuerst alle Semaphortickets reserviert, dann die Grösse und der Inhalt auf Leer geändert. Der Server fragt danach wieder den Client an, um den Inhalt zu erhalten. Sobald der Client den Inhalt generiert hat und zurückgegeben wurde, speichert der Server die Werte im Feld Content. Danach werden die Semaphortickets zurückgegeben. Solange UPDATE aktiv ist, kann kein READ, DELETE oder UPDATE durchgeführt werden.

3.1.5 Implementation DELETE

Falls bei einem Löschvorgang der Name auf NULL gesetzt wird, erhält das Array einen zweiten Punkt, wo die Stringlänge ebenfalls gleich 0 ist. Daraus folgt, dass das Array nicht mehr komplett durchgelaufen werden kann. Daher wird der Name nur auf "EMPTY"gesetzt. So ist garantiert, dass das Array immer komplett durchgelaufen werden kann. Die restlichen Einträge werden auf NULL, 0 oder leer gesetzt.

3.1.6 Implementation Semaphor

Im Array enthält jeder Eintrag eine Variable semval. Diese beinhaltet die maximalen Tickets für eine Datei. In den einzelnen Befehlsbereichen wird bei der Ausführung der Wert ausgelesen und per semctl in den Semaphor geladen. Danach holt sich der Prozess ein oder alle Tickets per semop. Sobald der kritische Bereich ausgeführt ist, wird das reservierte Ticket frei gegeben.

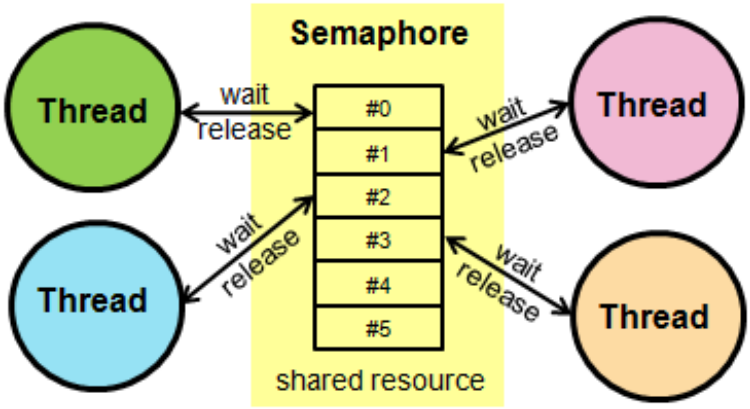

Abbildung 3.1: Semaphor sperrt Ressource für Threads[1]

3.2 Schwierigkeiten

In diesem Abschnitt möchte ich auf einige Probleme eingehen, die bei der Erstellung des Servers aufgetreten sind. Im Anschluss zu den jeweiligen Problemen wird meine Lösung angegeben.

3.2.1 Struktur Datei

In der Struktur Datei werden die Informationen zu den Dateien gespeichert. Hier gab es 2 grosse Entscheidungen bei der Implementation. Das erste Problem bestand in der Definition von der Variable Name und Content. Sind diese als undefinierte Arrays von Chars

[1]Quelle: `http://www.mbed.org`

deklariert, zeigt der nächste Eintrag im Strukturarray auf den gleichen Speicher, wie der vordere. So entstand das Problem, dass ein zweiter Eintrag den Namen des ersten Eintrages änderte. Dies ging so weiter für jeden weiteren Eintrag in der Struktur.

Das zweite Problem lag beim Durchsuchen. Der Name im Array wurde als Referenz genutzt. Der Durchlauf per Schlaufe lief mit der Prüfung auf NULL aber nur schlecht durch. Auch funktionierte er teilweise nicht, so dass das Programm hängen blieb und nicht mehr reagieren konnte.

Lösung

Problem 1 kam aus der Eigenschaft des undefiniertem Array. Der Speicher des Namensvariable zeigt für alle Einträge im Strukturarray auf die gleiche Adresse. So entstand die Eigenschaft, dass jeder weitere Eintrag den Namen auf die vorhergehenden Einträge vererbte. Die Lösung war, dass der Name nicht mehr eine undefinierte Länge besitzt, sondern das Char-Array von Beginn an auf eine bestimmte Grösse definiert wurde.

Problem 2 lies den Server an einer bestimmten Stelle einfrieren. Dieser Effekt war aber nicht voraussagbar, da er nur sporadisch auftrat. Die Änderung der while-Schleife auf die Prüfung der Stringlänge behob dieses Problem.

3.2.2 Generierung Messages

Kapitel 4

Fazit