Concurrent Virtual Filesystem Server

Roger Knecht

26. Mai 2014

Seminararbeit im Modul

„Concurrent C Programming“
im 6. Semester

Dozent: Nico Schottelius

Abgabetermin: 20. Juni 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Zusammenfassung

Im Rahmen der Seminararbeit für das Modul *Concurrent C Programming* an der *Zürcher Fachhochschule für Angewandte Wissenschaften* (ZHAW) soll eine Multi-User Applikation entwickelt werden. Ziel ist es, die Probleme von simultanen Speicherzugriffen mit der Programmiersprache C unter Linux zu erkennen und zu lösen. In dieser Arbeit soll ein Fileserver realisiert werden, der mehreren Benutzer den zeitgleichen Zugriff auf ein virtuelles Dateisystem erlaubt.

1.2 Anforderungen

Der Fileserver soll per TCP-Socket erreichbar sein und mehreren Client-Applikationen gleichzeitige Dateioperationen erlauben. Die Architektur soll auf dem Client/Server-Modell basieren und es soll kein globales Locking zum Einsatz kommen. Das Locking muss granularer sein, so dass zwei Dateien unabhängig voneinander editiert werden können. Mögliche Operationen sind erstellen, lesen, schreiben, löschen und auflisten von Dateien und Ordnern. Die genaue Funktionsweise der einzelnen Operationen ist im Netzwerkprotokoll festgehalten.

1.3 Gundprotokoll

Das Protokoll[1] basiert auf den Grundoperationen Erstellen, Lesen, Schreiben, Löschen und Auflisten.

Befehl	**Client**	**Server**
List	Client sendet: LIST	Server antwortet: ACK NUM FILES FILENAME FILENAME FILENAME ...
Create	Client sendet: CREATE FILENAME LENGTH CONTENT	Server antwortet: FILEEXISTS oder FILECREATED
Read	Client sendet: READ FILENAME	Server antwortet: NOSUCHFILE oder FILECONTENT FILENAME LENGTH CONTENT
Update	Client sendet: UPDATE FILENAME LENGTH CONTENT	Server antwortet: NOSUCHFILE oder UPDATED
Delete	Client sendet: DELETE FILENAME	Server antwortet: NOSUCHFILE oder DELETED

Tabelle 1: Grundprotokoll

[1] `https://raw.githubusercontent.com/telmich/zhaw_seminar_concurrent_c_programming/master/protokoll/fileserver`

2 Konzept

2.1 Erweiterung des Grundprotokolls

Die Anforderung an das Protokoll geht von einer flachen Dateistruktur aus. Das heisst es werden keine Unterordner erlaubt. Um die Aufgabe interessanter und herausfordernder zu gestalten, wird das Protokoll in dieser Arbeit um Unterordner erweitert. Diese Erweiterung verdeutlicht besonders gut die Komplexität von Deadlocks, Race-Conditions und Shared-Memory.

2.1.1 Anwendungsfälle

Die Anforderung ist es, eine Applikation zu entwickeln, die mehreren Benutzern gleichzeitig Operationen auf dem virtuellen Dateisystem ermöglichen. Das System muss mehrere Verbindungen aufbauen und Befehle miteinander ausführen. Falls Befehle auf dieselbe Ressource zugreifen, muss das System die Resource synchronisieren, damit es nicht zu einer Race Condition (siehe Abschnitt **3.3 Race-Conditions**) kommt.

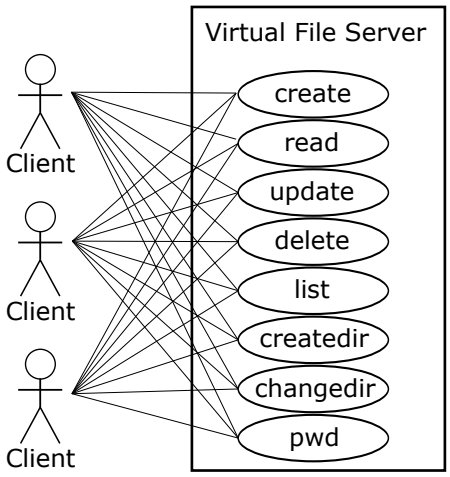

Abbildung 1: Anwendungsfälle

Die Benutzer können auf dem System Dateien erstellen (create), Dateien lesen (read), Dateien schreiben (update), Ordner erstellen (createdir), Dateien und Ordner löschen (delete), den aktuellen Arbeitsordner wechseln (changedir) und den Pfad zum aktuellen Arbeitsordner anzeigen (pwd).

2.1.2 Erweitertes Protokoll

Befehl	**Client**	**Server**
List ls	Client sendet: LIST	Server antwortet: ACK NUM FILES FILENAME FILENAME FILENAME ...
Create	Client sendet: CREATE FILENAME LENGTH CONTENT	Server antwortet: FILEEXISTS oder FILECREATED
Read cat	Client sendet: READ FILENAME	Server antwortet: NOSUCHFILE oder FILECONTENT FILENAME LENGTH CONTENT

Update	Client sendet: UPDATE FILENAME LENGTH CONTENT	Server antwortet: NOSUCHFILE oder UPDATED
Delete rm	Client sendet: DELETE FILENAME	Server antwortet: NOSUCHFILE oder DELETED
Createdir mkdir	Client sendet: CREATEDIR DIRNAME	Server antwortet: FILEEXITS oder DIRCREATED
Changedir cd	Client sendet: CHANGEDIR DIRNAME	Server antwortet: NOSOCHFILE oder DIRCHANGED
Pwd	Client sendet: PWD	Server antwortet: DIRECTORY

Tabelle 2: Erweitertes Protokoll

2.2 Abstraktionslayer

Die Software muss viele verschiende Aufgaben lösen. Das Lesen der Befehlszeilenargumenten, das Aufsetzen des Serversockets, das Erstellen von Threads, die Interpretation des Protokolls, die Verwaltung des virtuellen Dateisystems und das Lösen von Synchronisierungsprobleme.

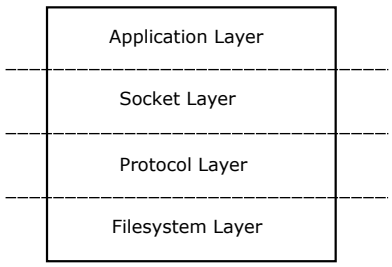

Abbildung 2: Abstraktionslayer

Um diese Komplexität zu bewältigen wird die Applikation in vier Ebenen (engl. Layer) unterteilt, wie in Abbildung 2 dargestellt. Jede dieser Ebenen ist genau für einen Bereich zuständig. Desweiteren darf eine Ebene maximal mit zwei anderen Ebenen verbunden sein, einer höheren und einer tieferen Ebene. Jede Ebene bietet der höheren Ebene eine eigene Schnittstelle (engl. Interface) an. Diese Regel verhindert zirkuläre Referenzierungen der Ebenen und verbessert die Qualität des Codes.

Application-Layer
Liest und interpretiert die Befehlszeilenargumente und startet mit entsprechenden Optionen den Socket.

Socket-Layer
Kümmert sich um die Kommunikationsverbindung mit den Clients. Pro Client wird eigener Thread erstellt.

Protocol-Layer
Interpretiert die Befehle des Clients und führt die Aktionen auf dem daraunterliegenden virtuellen Filesystem aus.

Filesystem-Layer
: Implementation des virtuellen Filesystems. Es abstrahiert die Dateioperationen und sorgt dafür, dass keine Synchronisationsprobleme auftreten. Das Filesystem ist komplett threadsafe.

2.3 Modell des virtuellen Dateisystems

Das virtuelle Dateisystem kennt nur einen Datentyp, den Node. Ein Node ist entweder eine Datei oder ein Ordner. Der Unterschied zwischen den beiden besteht lediglich darin, dass eine Datei einen Inhalt und ein Ordner Referenzen auf seine untergeordneten Nodes hat.

Jeder Node bestizt je zwei Read-/Write-Locks. Ein Open-Lock (grün) und ein Modify-Lock (rot). Read/Write-Locks können mehrfach lesend geöffnet werden, aber immer nur einmal schreibend und dies auch nur, wenn kein Leseschutz vorhanden ist[2].

Um mit einem Node zu arbeiten, muss dieser zuerst geöffnet werden. Das Öffnen des Nodes öffnet einen Leseschutz auf dem Open-Lock (siehe Abbildung 3). Solange ein Open-Lock geöffnet ist, wird der benutzte Speicher nicht freigegeben. Alle Threads die auf einen Node referenzieren, müssen deshalb den Node öffnen. Es kann durchaus sein, dass mehrere Threads einen Leseschutz über längere Zeit auf einem Node aufrecht erhalten.

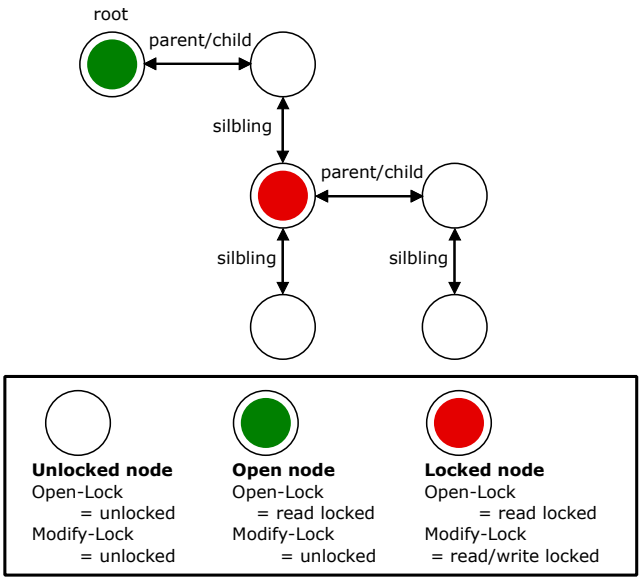

Abbildung 3: Virtuelles Dateisystem

Um Daten zu lesen oder zu schreiben kommt zusätzlich das Modify-Lock zum Einsatz. Dieses Lock wird allerdings nur kurze Zeit beansprucht. Eine Dateioperation wird den Schutz zu Beginn beanspruchen und nach Abschluss der Arbeit wieder freigeben.

Lesen einer Datei:

1. Leseschutz auf dem Open-Lock beanspruchen
2. Leseschutz auf dem Modify-Lock beanspruchen
3. Dateiinhalt lesen
4. Leseschutz auf dem Modify-Lock freigeben
5. Leseschutz auf dem Open-Lock freigeben

Schreiben einer Datei:

1. Leseschutz auf dem Open-Lock beanspruchen
2. Schreibschutz auf dem Modify-Lock beanspruchen
3. Dateiinhalt schreiben
4. Schreibschutz auf dem Modify-Lock freigeben
5. Leseschutz auf dem Open-Lock freigeben

[2] `http://en.wikipedia.org/wiki/Read/write_lock`

2.4 Threads vs. Prozesse

Die parallele Verarbeitung kann entweder mittels Threads oder über separate Prozesse realisiert werden. Beide Varianten bieten sowohl Vorteile wie auch Nachteile. Nachfolgend werden diese Vor- und Nachteile erläutert.

Threads

Ein Prozess mit mehreren Threads ist einfacher zu benutzen. Der gesamte Heap teilt sich die Threads. Das heisst, jeder Thread kann auf die Speicherbereiche des anderen zugreifen. Das bietet den Vorteil, dass keine zusätzliche Logik implementiert werden muss um Daten gemeinsam zu nutzen. Für die Speicherverwaltung können daher die Funktionen *malloc()* und *free()* aus der standard C-Library verwendet werden. Der Nachteil dieser Variante liegt in der Stabilität und der Sicherheit. Falls einer der Threads abstürtzt, beendet er den Serverprozess mitsamt allen Threads. Über benutzerdefinierte Signalhandler kann dieses Verhalt auch übersteuert werden. Es erfordert jedoch einen Mehraufwand gegenüber Prozessen. Da alle Threads den Speicher gemeinsam nutzen, können auch Speicherbereiche überschrieben werden, die ein Thread nur für sich beansprucht. Das birgt gewisse Sicherheitsrisiken.

Prozesse

Prozesse teilen grundsätzlich keine Speicherbereiche miteinander. Um Daten zwischen Prozessen zu teilen gibt es unterschiedliche Arten von *Interprocess Communitation (IPC)*. Für diese Aufgabe würde Shared-Memory in Frage kommen. Dabei können spezifische Speicherbreiche explizit zwischen mehreren Prozessen geteilt werden. Diese Methode eignet sich allerdings nicht für kleine Allokationen. Speicher der nicht freigegeben wird kann von anderen Prozessen nicht verändert werden. Das sorgt für mehr Stabilität und Sicherheit. Der grosse Nachteil ist, dass man eine eigene Speicherverwaltung für den Shared-Memory Bereich schreiben muss.

In dieser Arbeit werden Threads verwendet, weil die Speicherverwaltung wesentlich einfacher ist als mit Shared-Memory. Für das Shared-Memory hätten eigene *malloc()/free()* Funktionen implementiert werden müssen. Das hätte die Komplexität des Codes erhöht und wäre vom Zeitumfang nicht möglich gewesen. Als Bibliothek kommt hierfür die Pthread Library zum Einsatz. Das ermöglicht eine gute Portierbarkeit auf andere POSIX kompatible Systeme. Pthread liefert ausserdem eine Fülle an Synchronisierungsmechanismen wie Semaphoren, Mutexe und Barrieren.

3 Problemstellung

Bei der Entwicklung dieser Applikation müssen verschiedene Aspekte berücksichtigt werden, die zu einem Fehlerverhalten oder sogar zum Absturz führen. In diesem Abschnitt werden die Problemstellungen skizziert und mögliche Lösungsansätze kurz geschildert. Im Abschnitt **4 Realisierung** werden die gewählten Lösungen detailiert behandelt.

3.1 Memory-Leaks

Wenn mehrere Threads bzw. Prozesse Arbeitsspeicher teilen möchten, muss hierfür Speicher auf dem Heap belegt werden. In der Programmiersprache C gibt es keine automatische Speicherverwaltung und der Programmierer muss sich selber um die Freigabe der belegten Speicherbereiche kümmern.

Wird ein allozierter Speicherbereich nicht freigegeben und es gibt im Programm keine Referenzen mehr auf diese Speicherstelle, so entsteht ein Memory-Leak. Dieser Speicherbereich kann nicht mehr neu alloziert werden, da er als belegt markiert ist und die Applikation die Speicherstelle nicht kennt um diesen weiterhin zu benutzen oder freizugeben. Der Bereich ist also nicht mehr benutzbar (Abbildung 4).

Um Memory-Leaks zu verhindern, ist es ratsam ein Konzept aufzustellen, wann Speicher alloziert und wann freigegeben wird. Eine andere Möglichkeit besteht darin Smartpointers zu verwenden[3].

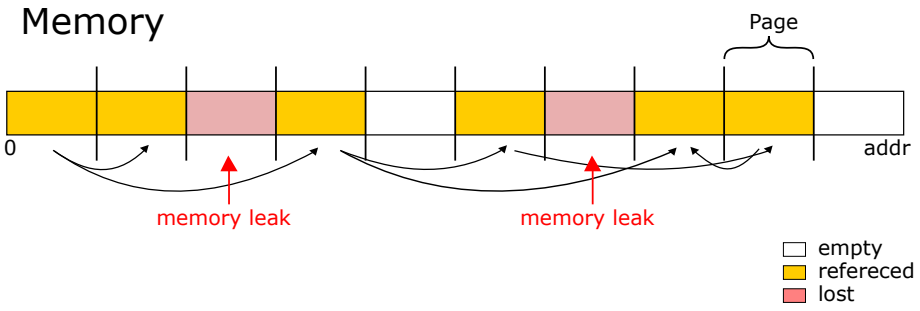

Abbildung 4: Nicht referenzierte Speicherbereiche resultieren in Memory-Leaks

3.2 Segmentation-Fault

Die Programmiersprache C erlaubt es auf eine beliebige Speicheradresse zuzugereifen. Falls auf eine Speicheradresse zugegriffen wird, die nicht dem Prozess zugewiesen ist, kommt es zu einem Segmentation-Fault (Abbildung 5). Dabei sendet das Betriebssystem ein Signal und das Programm beendet sich standardgemäss. Noch schlimmer ist es jedoch wenn fälschlicherweise auf ein Speicherbereich zugegriffen wird, der bereits dem Prozess zugewiesen ist. Denn das bemerkt das Betriebssystem nicht und das Programm wird weiter ausgeführt. Dabei werden Daten überschrieben und das Programm befindet sich in einem inkonsisten Zustand. Dieses Verhalten kann unter Umständen sehr schwer zu debuggen sein, weil sich die Fehler erst viel später zeigen.

[3] `http://en.wikipedia.org/wiki/Smart_pointer`

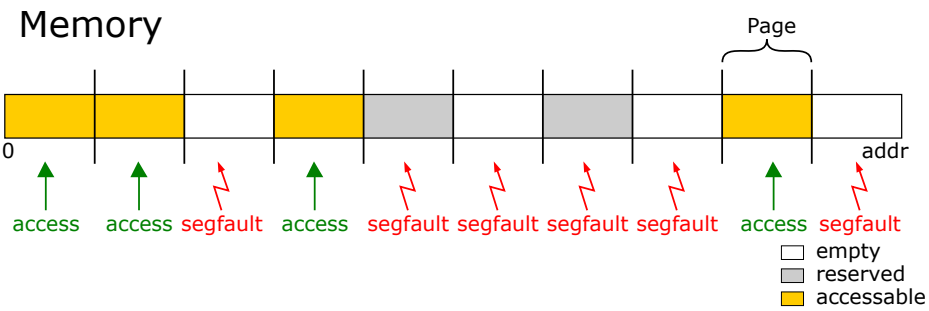

Abbildung 5: Beim Zugriff auf nicht alloziertem Speicher kommt es zu einem *Segmentation-Fault*

3.3 Race-Conditions

Ein Fall, in dem der Ausgang eines Programms davon abhängig ist, welcher Thread bzw. Prozess zeitlich zuerst ausgeführt wird, nennt man Race-Condition. Das ist insbesondere der Fall, wenn zwei Threads gleichzeitig auf eine Ressource zugreifen. Deshalb ist es notwendig geteilte Ressourcen zu synchronisieren. Dadurch kann nur ein Thread auf einmal auf diese Ressource zugreifen.

3.4 Deadlock

Blockieren sich zwei Threads gegenseitig so, dass beide warten bis der jeweils andere weiterfährt, kommt es zu einem Stillstand. Dieser Zustand wird Deadlock genannt. Um Deadlocks zu vermeiden, muss eine Konvention erstellt werden, die besagt in welcher Reihenfolge Ressourcen gelockt werden. Für diese Applikation wird im *Filesystem Layer* nach der Reihenfolge, die in Abbildung 6 gezeigt wird, gelockt.

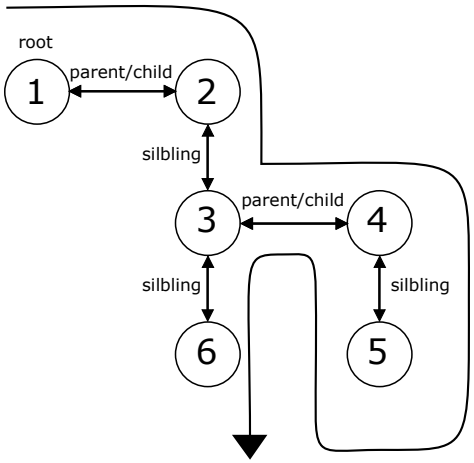

Abbildung 6: Um Deadlocks zu vermeiden muss beim Sperren von Nodes diese Reihenfolge eingehalten werden

4 Realisierung

Die Software ist so implementiert, dass sich nur die unterste Ebene um die Synchronisierung von Threads kümmern muss. Ebenfalls wird die Speicherverwaltung für das Dateisystem auf dieser Ebenen gelöst. Deshalb müssen die höhren Ebenen keine Logik zur Synchronisierung einbauen.

4.1 Memory-Handling

Um Memory-Leaks und Segmentation-Faults zu vermeiden, gibt es eine strenge Vorschrift wann und wo Speicher alloziert bzw. wieder fregegeben werden darf.

4.1.1 Node Initialisieren

Die Methode *vfs_create()* in *src/vfs.h* ist der einzige Ort, an dem Speicher für einen Node angelegt wird.

4.1.2 Node Freigabe

Es gibt die Methode *vfs_delete()*, die einen Node löscht. Allerdings wird dabei der Speicher nicht freigegeben. Es könnte sein, dass noch weitere Threads eine Referenz auf genau diesen Node haben. Eine Freigabe dieses verwendeten Speichers könnte einen Segmentation-Fault zur Ursache haben oder noch schlimmer, ein undefiniertes Verhalten. Deshalb wird der Node dabei nur als gelöscht markiert.

Erst die Methode *vfs_close()* gibt den Speicher frei und zwar nur dann, wenn keine Referenzen auf den Node existieren.

4.1.3 Initialisierung des Dateiinhalts

Beim Erstellen eines Nodes wird noch kein Speicher für den Dateiinhalt angelegt. Nur die Methode *cfs_write()* alloziert Speicher für den Inhalt. Falls zuvor bereits Speicher dafür angelegt wurde, wird dieser zuerst freigegeben.

4.2 Synchronisation

Alle Nodes besitzen je zwei Read/Write-Locks, damit nicht mehrere Threads gleichzeitig Daten modifizieren. Dadurch können Race-Conditions vermieden werden. Um Deadlocks zu verhindern, gibt es eine strenge Reihenfolge, wie Nodes gelockt werden dürfen (siehe Abschnitt 3.4).

4.2.1 Node Erstellen

Beim Erstellen eines Nodes wird zuerst der Parent-Node gelockt und danach der neue Node als Child hinzugefügt. Dabei muss darauf geachtet werden, dass nicht bereits ein Node mit demselben Namen existiert. Abbildung 7 zeigt ein einfaches Beispiel ohne Child-Nodes.

Schritte:

- Parent-Node locken
- Durch alle Childs iterieren. Mit Fehlermeldung abbrechen, falls ein Node mit dem selben Namen existiert
- Node am Ende hinzufügen

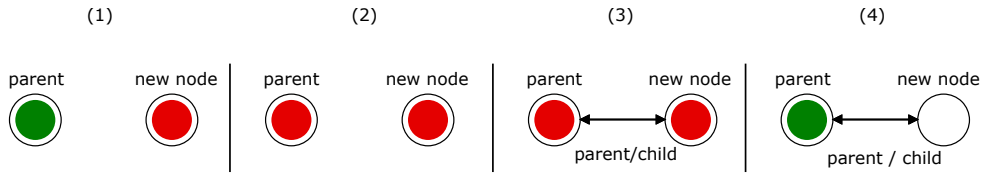

Abbildung 7: Ausgangslage (1), Parent sperren (2), Nodes verknüpfen (3), Freigabe der Locks (4)

4.2.2 Node Auflisten

Das Auflisten aller Nodes funktioniert ähnlich dem Erstellen. Zuerst wird der Parent-Node gelockt. Anschliessend wird über alle Childs gesprungen und dabei deren Namen gelesen, wie Abbildung 8 zeigt.

Schritte:

- Parent-Node locken
- Durch alle Childs itaerieren und deren Namen auslesen.

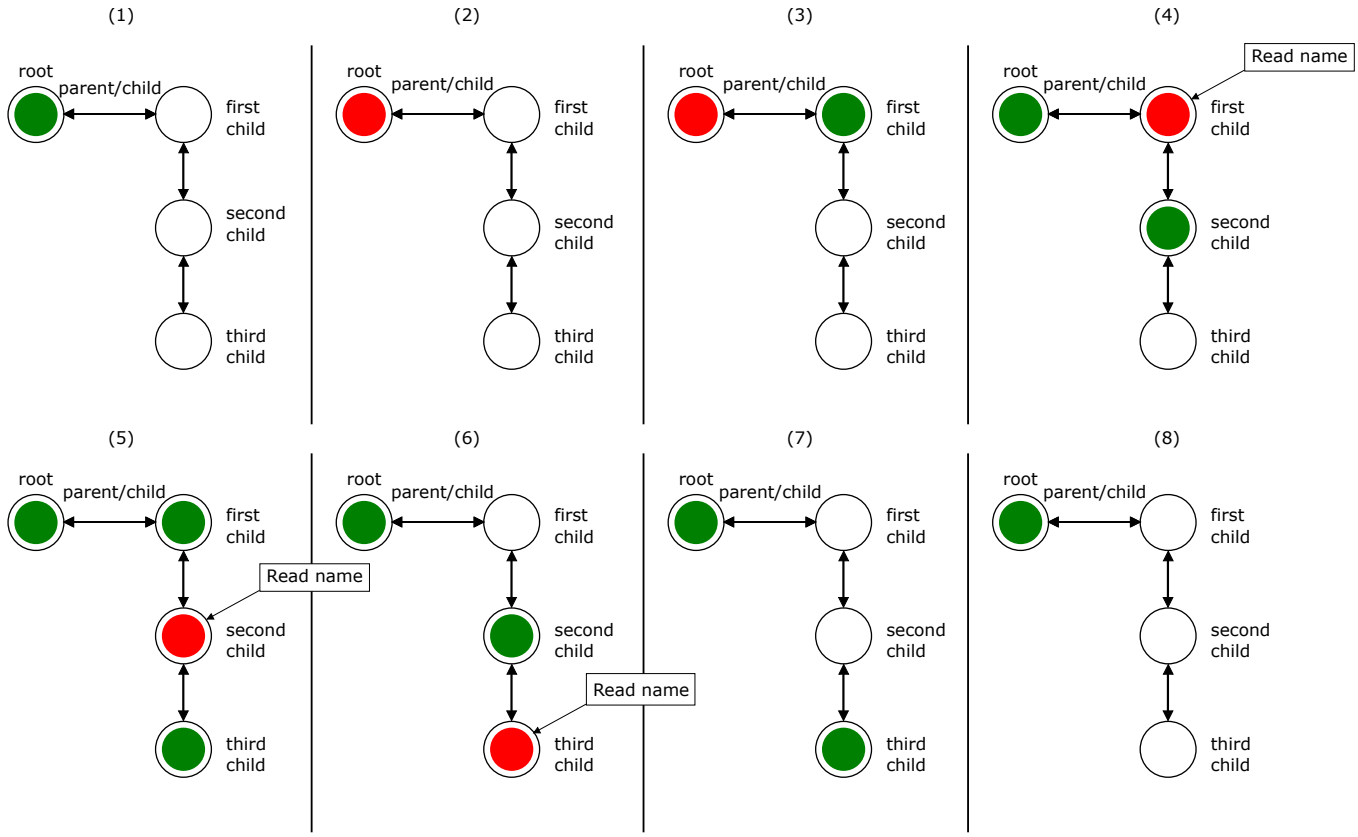

Abbildung 8: Ausgangslage (1), Parent sperren (2), erstes Child öffnen (3), Namen des ersten Child lesen (4), Namen des zweiten Child lesen (5), Namen des dritten Child lesen (6), Locks freigeben (7), Anfangszustand (8)

4.2.3 Node Lesen

Beim Lesen des Dateiinhalts oder der Dateieigenschaften, wird kurzzeitig ein Read-Lock gesetzt (Abbildung 9).

Schritte:
- Read-Lock setzen
- Node auslesen
- Read-Lock freigeben

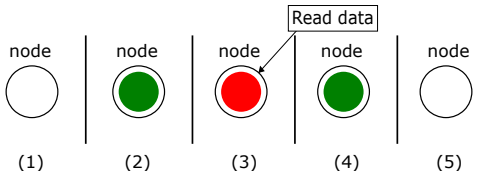

Abbildung 9: Ausgangslage (1), Datei öffnen (2), Read-Lock Datei (3), Read-Lock schliessen (4), Datei schliesen (5)

4.2.4 Node Schreiben

Das Schreiben des Dateiinhaltes oder der Dateieigentschaft ist identisch zum Lesen. Mit dem Unterschied, dass ein Write-Lock gesetzt wird (Abbildung 10).

Schritte:
- Write-Lock setzen
- Node schreiben
- Write-Lock freigeben

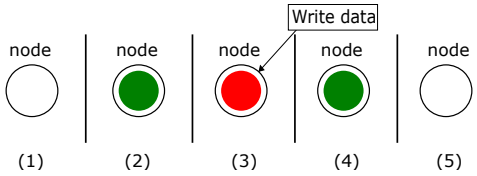

Abbildung 10: Ausgangslage (1), Datei öffnen (2), Write-Lock Datei (3), Write-Lock schliessen (4), Datei schliessen (5)

4.2.5 Node Löschen

Wie bereits vorgängig erwähnt, besteht das Löschen nur im Markieren eines Nodes als "gelöscht". Das eigentliche Freigeben des Speichers passiert beim Schliessen der Datei. Deshalb ist das Löschen nur das Schreiben einer Dateieigentschaft. Um möglichst bald alle Referenzen auf diesen Node zu beseitigen, wird der Node *abgekoppelt* wie der Grafik 11 zu entnehmen ist.

Schritte:

- Vorgänger öffnen (Parent oder previous silbling)
- Vorgänger mit Write-Lock sperren
- Den zu löschenden Node mit Write-Lock sperren
- Nachfolgender Node öffnen (falls vorhanden)
- Nachfolgender Node mit Write-Lock sperren (falls vorhanden)
- Node als gelöscht markieren und Nodes neu verknüpfen
- Write-Locks freigeben
- Nodes schliessen

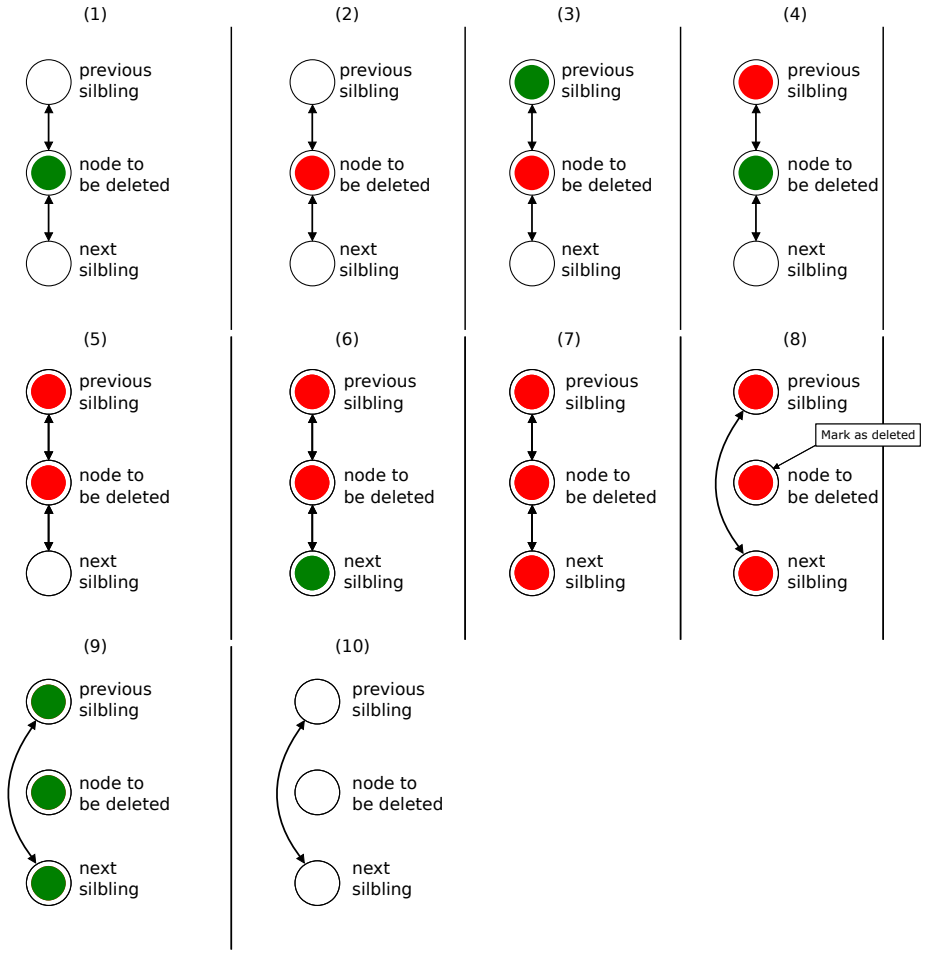

Abbildung 11: Ausgangslage (1), Read-Lock auf dem zu löschenden Node (2), wechsel zum Vorgänger um die Lock-Reihenfolge einzuhalten (3), Vorgänger mit Write-Lock sperren (4), zu löschenden Node mit Write-Lock sperren (5), Nachfolger mit Write-Lock sperren (6/7), Node als gelöscht markieren und die anderen beiden Node neu verknüpfen (8), Locks freigeben (9/10)

5 Benutzeranleitung

5.1 Server Kompilieren

Den Server kann man über das Makefile kompilieren:

```
1 $ make
```

5.2 Server Starten/Stoppen

Ruft man die Server-Executable ohne Argumente auf, so erscheint die Usage.

```
1 $ ./fileserver
2 usage: fileserver -p port [-c maxclients]
3    [-l trace|debug|info|warn|error]
```

Der Port muss zwingend angegeben werden. Optional kann auch die maximale Anzahl offener Verbindung angegeben werden. Standardmässig ist der Wert auf 50 parallelen Verbindungen eingestellt. Über das Argument *-l* kann das Log-Level variert werden. Ohne diese Angabe ist das Level auf *info* eingestellt.

Man startet den Server folgendermassen:

```
1 $ ./fileserver -p 8000
```

In der Aufgabenstellung ist vorgegeben, dass der Server auch wie folgt gestartet werden kann:

```
1 $ ./run
```

Deshalb wurde dieses Bash-Script erstellt, welches den Server auf Port 8000 startet. Beenden kann man den Server mit *Ctrl*+*C*.

5.3 Automatische Tests

Zuerst muss der Server über das Run-Script gestartet werden. Anschliessend können die automatischen Tests über folgendes Bash-Script gestartet werden:

```
1 $ ./run &
2 $ ./test
```

5.4 Telnet

Es ist auch möglich eine Verbindung manuell per Telnet aufzubauen.

```
1  $ ./run &
2  [1] 5771
3  $ telnet localhost 8000
4  Trying ::1...
5  Trying 127.0.0.1...
6  Connected to localhost.
7  Escape character is '^]'.
8  hello client and welcome to multithreading fileserver
9  > ls
10 > mkdir test
11 DIRCREATED Directory created
12 > ls
13 test
```

```
14 > exit
15 Connection closed by foreign host.
```

5.5 Fuse-Treiber

Am Ende der Arbeit wurde noch ein Fuse-Treiber entwickelt, um das virtuelle Dateisystem in das Betriebssystem zu integrieren. Dropbox und weitere Anbieter von Cloud-Storage verwenden diese Methode um Benutzern Zugriff auf ihre Daten zu geben.

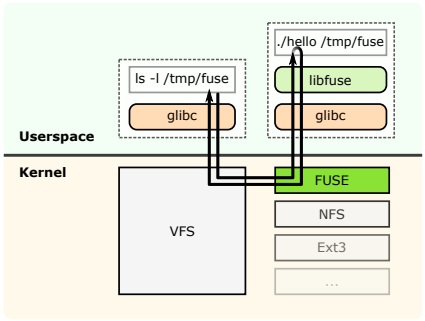

Abbildung 12: Architektur von Fuse (Quelle: Wikimedia Foundation)

Fuse (**F**ilesystem in **Use**rspace) ist ein Linux-Treiber um Dateisysteme im Userspace zu entwickeln. Der Treiber kann dabei in diversen Sprachen wie zum Beispiel C, Perl und Python entwickelt werden.

Wie bereits das Test-Skript ist auch der Fuse-Treiber in Python geschrieben. Die Funktionen des Fuse-API werden in Befehle umgewandelt und an den Fileserver gesendet. Die Rückmeldung wird interpretiert und Fuse zurückgegeben. Die Architektur von Fuse ist in Abbildung 12 abgebildet.

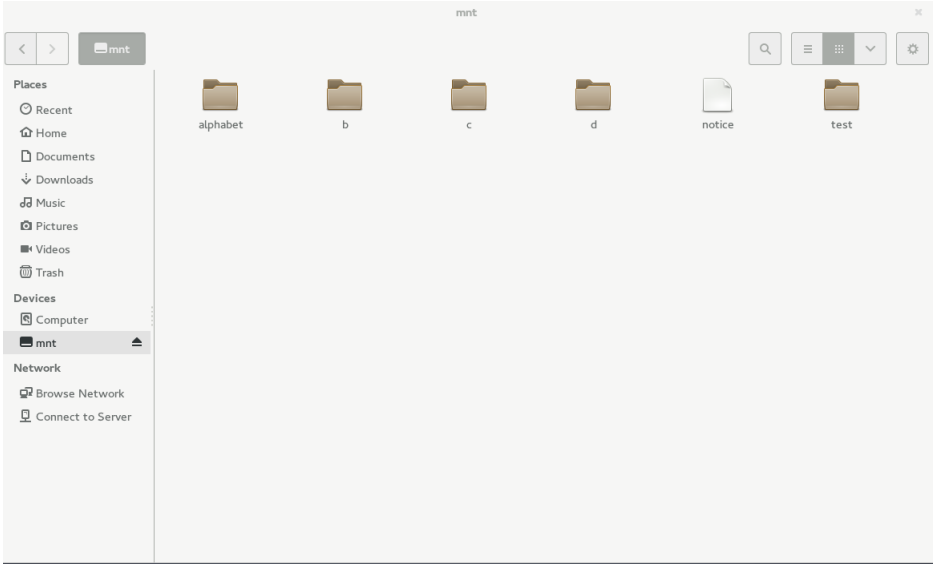

Abbildung 13: Nautilus Filebrowser greift über den Fuse-Treiber auf das VFS zu

A Literaturverzeichnis

[1] Brian W. Kernighan, Dennis M. Ritchie, *C Programming Language*, Prentice Hall, 2nd Edition, 1988

[2] W. Richard Stevens, Stephen A. Rago, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 3rd Edition, 2013

[3] Eduard Glatz, *Betriebssysteme - Grundlage, Konzepte, Systemprogrammierung*, dpunkt.verlag, 2nd Edition, 2010

[4] Heike Jurzik, *Debian GNU/Linux*, Galileo Press, 5th Edition, 2013

[5] Harald Maassen, *LPIC-1*, Galileo Press, 3rd Edition, 2013

[6] Andrew S. Tannenbaum, David J. Wetherall, *Computernetzwerke*, Pearson, 5th Edition, 2012

[7] Andrew S. Tannenbaum, Maarten van Steen, *Verteilte Systeme*, Pearson, 3rd Edition, 2008

[8] W. Richard Stevens, *TCP/IP*, VDE Verlag, 1st Edition, 2010

[9] Bernd Oestereich, *Die UML- Kurzreferenz für die Praxis*, Oldenbourg Wiss, 4th Edition, 2001

[10] Erich Gamma, Richard Helm, Ralph Johson, John Vlissides, Dirk Riehle, *Entwurfsmuster*, Addison-Wesley, 1st Edition, 2004

[11] Gustav Pomberger, Heinz Dobler, *Algorithmen und Datenstrukturen*, Pearson, 1st Edition, 2008

[12] Mark Lutz, *Programming Python*, O'Reilly Media, 4th Edition, 2011

[13] FUSE Community, *Offizielle Webseite von FUSE*, `http://fuse.sourceforge.net`

[14] FUSE Community, *FUSE Python tutorial*, `http://sourceforge.net/apps/mediawiki/fuse/?title=FUSE_Python_tutorial`

B Index