FILE SERVER

CONCURRENT PROGRAMMING IN C

ZÜRCHER HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN
STUDENT: RAFFAEL SANTSCHI
BETREUER: NICO SCHOTTELIUS
ABGABE: 22.06.2014

INHALTSVERZEICHNIS

EINLEITUNG

AUSGANGSLAGE

Es soll ein File-Server in C programmiert werden. Dieser soll mittels vorgegebenen Protokoll Files erstellen, bearbeiten, lesen und löschen können. Zusätzlich soll auch eine Liste aller Files angezeigt werden können.

LERNINHALTE

Selbstständige Definition des Funktionsumfangs des Programmes unter Berücksichtigung der verfügbaren Ressourcen im Seminar.
Konzeption und Entwicklung eines Programms, das gleichzeitig auf einen Speicherbereich zugreift.
Die Implementation erfolgt mithilfe von Threads oder Forks und Shared Memory (SHM).

VORGABEN

- Dateien sind nur im Speicher vorhanden
- Das echte Dateisystem darf NICHT benutzt werden
- Mehrere gleichzeitige Clients
- Lock auf Dateiebene (kein Globaler Lock)
- Debug-Ausgaben von Client/Server auf stderr

PROTOKOLL

List

```
Client sendet:
    LIST\n

Server antwortet:
    ACK NUM_FILES\n
    FILENAME\n
    FILENAME\n
    FILENAME\n
    ...

Server Beispiel:
    ACK 3
    abc
    def
    aei
```

Create

```
Client sendet:
    CREATE FILENAME LENGTH\n
    CONTENT

Client Beispiel:
    CREATE abc 6\n
    Hello\n
```

```
Server antwortet:
    FILEEXISTS\n
oder
    FILECREATED\n
```

Read

```
Client sendet:
    READ FILENAME\n

Client Beispiel:
    READ abc\n

Server antwortet:
    NOSUCHFILE\n
oder
    FILECONTENT FILENAME LENGTH\n
    CONTENT

Server Beispiel:
    FILECONTENT abc 5\n
    1234\n
```

Update

```
Client sendet:
    UPDATE FILENAME LENGTH\n
    CONTENT

Client Beispiel:
    UPDATE abc 3\n
    12\n

Server antwortet:
    NOSUCHFILE\n
oder
    UPDATED\n
```

Delete

```
Client sendet:
    DELETE FILENAME\n

Client Beispiel:
    DELETE abc\n

Server antwortet:
    NOSUCHFILE\n
oder
    DELETED\n
```

MANUAL

BENUTZUNG

START

Wenn man im Hauptverzeichnis ist, gibt es ein Makefile. Man kann mit „make“ das run Executable und test Executable erstellen lassen. Danach kann man mit „./run“ den Server starten, es gibt auch die Möglichkeit den Port vom Server zu konfigurieren „./run 8000“. Der Default Port ist 7000. Wenn der Server läuft, kann man den Test-Client starten „./test“.

BEENDEN

Der Test-Client und alle anderen Tests beenden sich selber. Der Server kann mittels „Ctrl+C“ bzw. Signal INT oder Signal USR1 beendet werden. Der Server beendet dann alle laufenden Thread und leert die Listen und beendet sich dann.

VERZEICHNISSTRUKTUR

Im Hauptverzeichnis gibt es zwei Ordner Client und Server. Im Client Verzeichnis ist ein Java Projekt drin, welches einen Client für den File-Server implementiert. Das Server Verzeichnis ist erneut unterteilt in folgende Verzeichnisse und Dateien:

- include (Alle header files)
-- …
- lib
-- file-linked-list.c (Concurrent Linked List für Files mit create, update, read, delete und list)
-- logger.c (Logging Funktionen finest, debug, info, warn und error)
-- regex-handle.c (Regex Funktionen compile und match)
-- serverlib.c (Error Handling)
-- thread-linked-list.c (Linked List für Threads mit add und cleanup)
-- transmission-protocols.c (Ein-/Auslesen von String über das TCP/IP und Protokoll handling)
-src
-- server.c (TCP/IP Verbindung aufbau und Worker Thread starten)
-tests
-- client-test.c (Client welcher über TCP/IP Files erstellt, ändert, liesst und löscht)
-- concurrent-load-test.c (Concurrent Tests ohne TCP/IP)
-- file-linked-list-test.c (Unit Tests für Concurrent Linked List)
-- load-test.c (Load Test für den Server, viele Lese/Schreib/Lösch Vorgänge produzieren)
-- logger-test.c (Unit Tests für Logger)
-- message-creator.c (Helfer Datei, um gemäss Protokol korrekte Messages zu schreiben)
-- server-test.c (Unit tests für CRUD und List Funktionen auch mit Fehlerfällen)
-- tester.c (Main Datei für CUNIT Tests)
-- thread-llinked-list-test.c (Unit Tests für Thread Linked List)

TESTING

Zuerst wurde ein Test Client in Java erstellt, da das Wissen in C noch nicht so fortgeschritten war. Bei dieser Entscheidung erhoffte man sich auch ein Vorteil, da in Java Multi-Threading sehr einfach zu erreichen ist. Der Test Client beherrscht das Protokoll und kommuniziert über TCP/IP. Mit dem Java Client konnten guten Last Tests gefahren werden, jedoch wahr es etwas mühsam hin und her zu switchen und deshalb wurde nach einer anderen Lösung gesucht. Man entschied sich für CUnit. Der Vorteil an CUnit war, dass auch einzelnen Funktionen in C getestet werden konnte. Es wurden Unit Tests für die einzelnen Funktionen erstellt und danach auch noch Multi-Threaded Testläufe. Dafür wurde zwei eigene kleine Test Client geschrieben (concurrent-load-test.c und client-test.c), diese verfügen über eine main-Methode, somit kann man sie ohne Framework starten. Sie führen diverse Aufrufe auf dem Server aus, wobei jeder einzelne Aufruf in einem eigenen Thread. Die Tests kann man mittels „make test-concurrent“ oder „make && ../test“ laufen lassen.

JAVA TEST CLIENT

Der Java Test Client hat mehrere Test Methoden. Die spannendsten sind „concurrentTestWith50DiffPrefixes()“, „concurrentTest()“ und „painConcurrentTest()“ in der Java Klasse ConcurrentTest. Die Tests laufen mit einem Tester-Thread, welcher zwei Files erstellt, diese geändert, danach wieder liesst und dann eine Liste aller Files holt und seine zwei Files wieder löscht. Beim „concurrentTestWith50DiffPrefixes()“-Test werden 50 Threads mit verschiedenen Filenamen gestartet um zu schauen, ob der Server mit viel Last umgehen kann. Im „concurrentTest()“- und „painConcurrentTest()“-Test werden 50 bzw. 200 Threads mit denselben Filenamen gestartet, mit diesem Tests wird überprüft, ob das Locking-Verfahren sauber funktioniert. Die Antworten vom Server werden durch einen selber geschriebenen Parser geschickt, der möglichst generisch ist und mit Reflection arbeitet, damit das Protokoll beliebig erweiterbar ist.

C TEST CLIENT

Der Client „client-test.c“ erstellt 100 Files, verändert sie, liesst sie aus, holt sich eine Liste der Files und löscht die 100 Files danach wieder. Jede Aktion beim Client läuft in einem eigenen Thread um die Concurrency zu testen. Die Messages, welche dem Server geschickt werden, werden in einem seperaten C-File „message-creator“ erstellt, da sie für andere Tests auch benötigt werden.

LÖSUNG

VERWENDUNG VON FREMDEM CODE

Einige Funktionen in transmission-protocols.c und serverlib.c wurden im Modul Systemsoftware von Karl Brodowsky zur Verfügung gestellt und in diesem Projekt wieder verwendet oder als Basis für eine Eigenentwicklung verwendet. Die Funktionen sind auf github und unter einer GPLv2 Lizenz.

ANSATZ

DYNAMISCHE LISTE

Schon früh war klar, dass für eine dynamische Liste von Files in C wahrscheinlich eine Linked List das Beste wäre, da man nicht wie in Java ein ArrayList hat, welche sich dynamisch vergrössert. Durch das Fach Algorithmen und Datenstrukturen war das Prinzip der Linked List bekannt und ein erster Entwurf stand ziemlich schnell. Es war stellte sich jedoch als eine herausfordernde Aufgabe heraus, die Linked List in eine Concurrent Linked List zu verwandeln. Dies wurde mit einem iterativen Vorgehen, sehr viel Zeit und Überlegungsarbeit gemeistert.

PROTOKOLL

Die Eingaben kommen mit einem vorgegebenen Schema, somit kann der Input durch einen Filter gelassen werden und erhält dann die nötigen Elemente. Doch wie man einen String in C parsen kann, war zu diesem Zeitpunkt nicht bekannt. Diverse Recherchen in Internet führten zu einer Implementation von Regex in C (Using regular expressions in C). Mit einer etwas speziellen Regex Notation konnten die nötigen Elemente aus dem String ausgelesen werden.

Regex Notation in C:
```
"UPDATE[[:blank:]]+([[:graph:]|[:blank:]]+)[[:blank:]]+([[:digit:]]+)[[:cntrl:]]+([
[:graph:]|[:blank:]]+)"
```
Regex Notation in Perl:
```
"UPDATE\s+([\x21-\x7E\s]+)\s+([\d]+)[\x00-\x1F\x7F]+([\x21-\x7E\s]+)"
(POSIX Bracket Expressions)
```

LOGGING

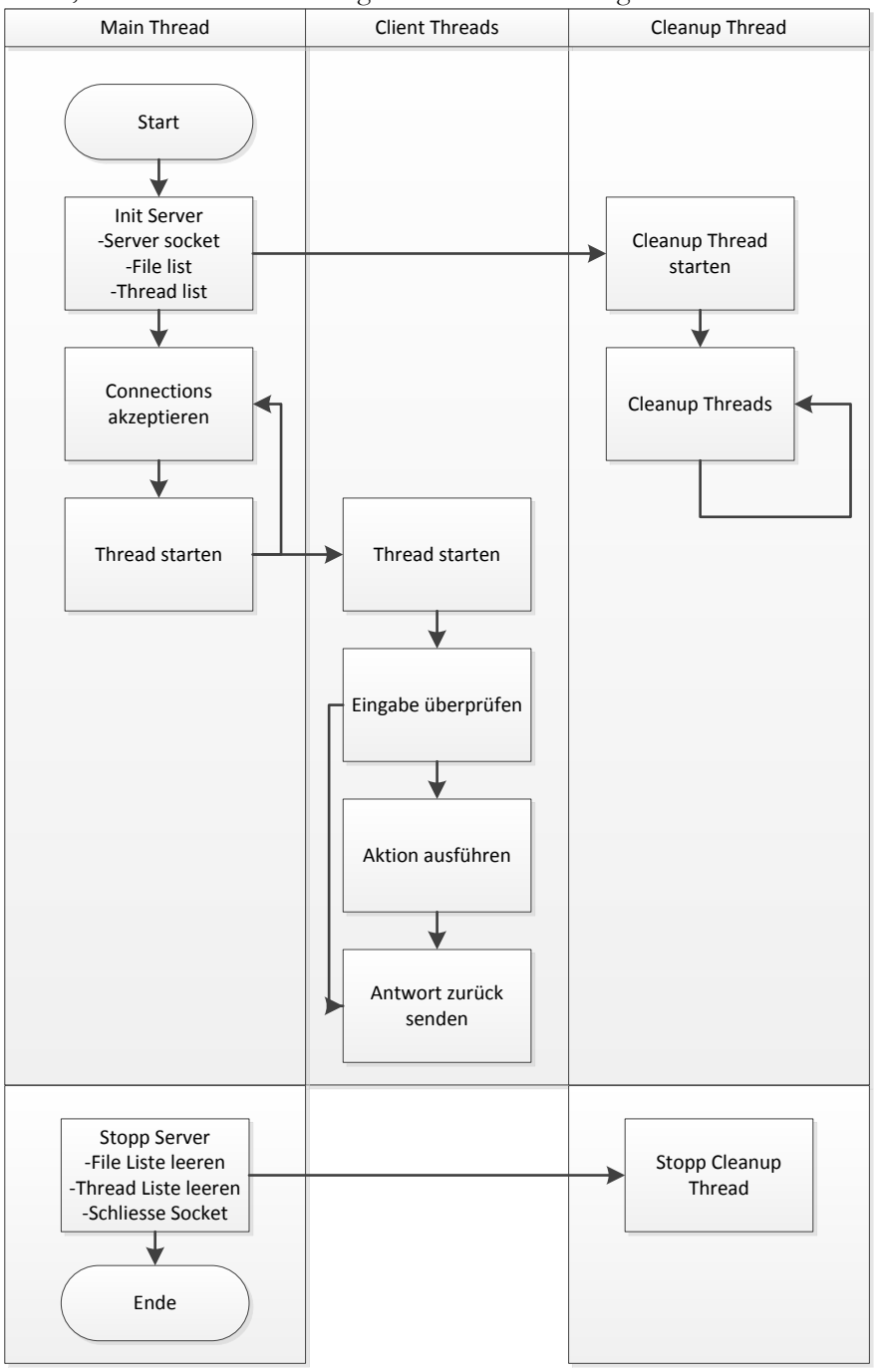
Eine Applikation braucht verschiedene Logging Stufen, welche man dann je nach Einsatz beliebig setzen kann. Es wurden zuerst die Stufen Debug, Info, Warning und Error implementiert, gegen den Schluss als die Ausgaben von allen Pointer unnötig wurde, wurde noch eine Stufe Finest hinzugefügt. Bei den Log Funktionen kann man beliebig viele Elemente mit geben, welche dann in das Template eingefügt werden, wie man dies zum Beispiel von printf kennt. Zur besseren Übersicht wurde noch eine Hierarchie-Stufe einbauen, damit nicht zu viel Zeit mit der Implementation verloren ging, wird nun die Hierarchie-Stufe mit einem Integer beim Aufruf der Logging-Funktion mitgegeben. Im folgenden Beispiel sieht man zwei Log Einträge mit verschiedener Hierarchie-Stufe und Log Level und den Code dazu:

140316902713152 Mon Jun 16 15:23:22 2014 [INFO]: Start 100 threads, which create a file
140365498795776 Mon Jun 16 14:26:38 2014 [DEBUG]: Try to connect to 127.0.0.1 on port 7000

```
info(0, "Start %d threads, which create a file", max_files_to_test);
debug(2, "Try to connect to %s on port %d\n", server_ip, server_port);
```

GESAMT ABLAUF

Das Main Programm initialisiert die Concurrent Linked List für die Files und die Thread liste und öffnet den Server Socket. Danach hört er nur noch auf dem Socket und eröffnet für jeden Request einen neuen Thread. Parallel zum Main Thread läuft der Cleanup Thread, welcher immer wieder die erstellen Threads einsammelt. Im neu erstellen Client Thread wird die Eingabe zuerst überprüfen, bei einer gültigen Eingabe wird die gewünschte Aktion ausgeführt und dann die Antwort an den Sender zurück geschickt. Beim Stoppen des Servers wird zuerst der Cleanup Thread beendet, dann werden die Listen geleert und der Socket geschlossen.

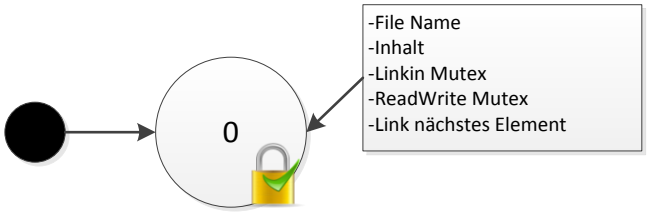


LOCKING

WIE WURDE DER GLOBALE LOCK UMGANGEN?

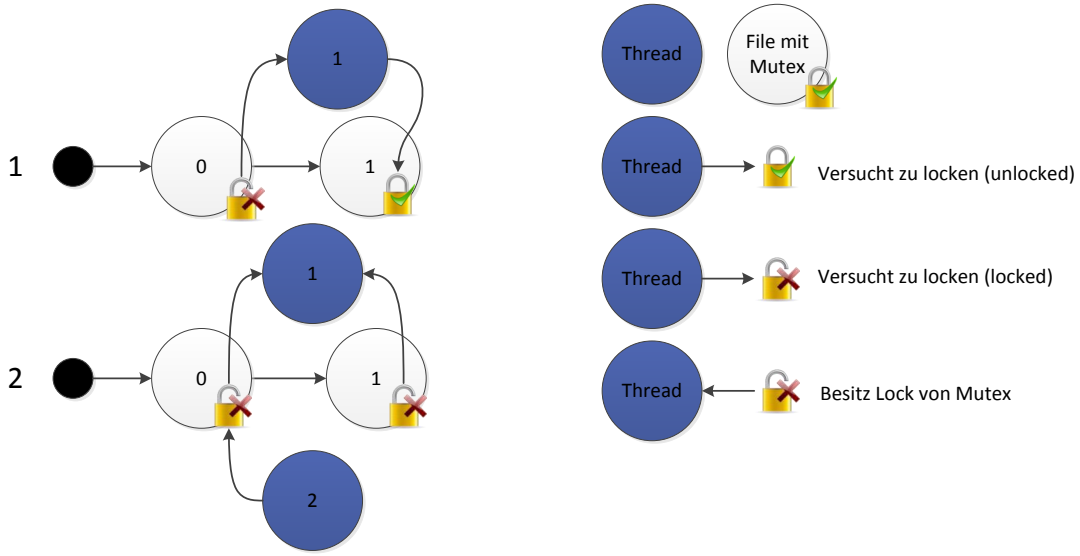
Eine der Vorgaben war, dass man kein globales Locking haben darf, um dies zu umgehen, wurde eine Concurrent Linked List erstellt. Jedes Element besitzt einen Mutex für das Linking und einen ReadWrite-Mutex für das Lesen und Schreiben. Der Linking Mutex wird gelockt, wenn auf das Element zu gegriffen wird und erst wieder freigegeben, wenn die Modifikation erfolgt ist.

Damit das Locking schon von Anfang an möglich ist, wird beim Initialisieren ein Element 0 eingefügt, welches bereits über einen Mutex verfügt. Die Elemente enthalten File Name, Inhalt, die beiden erwähnten Mutex und einen Link zum nächsten Element.

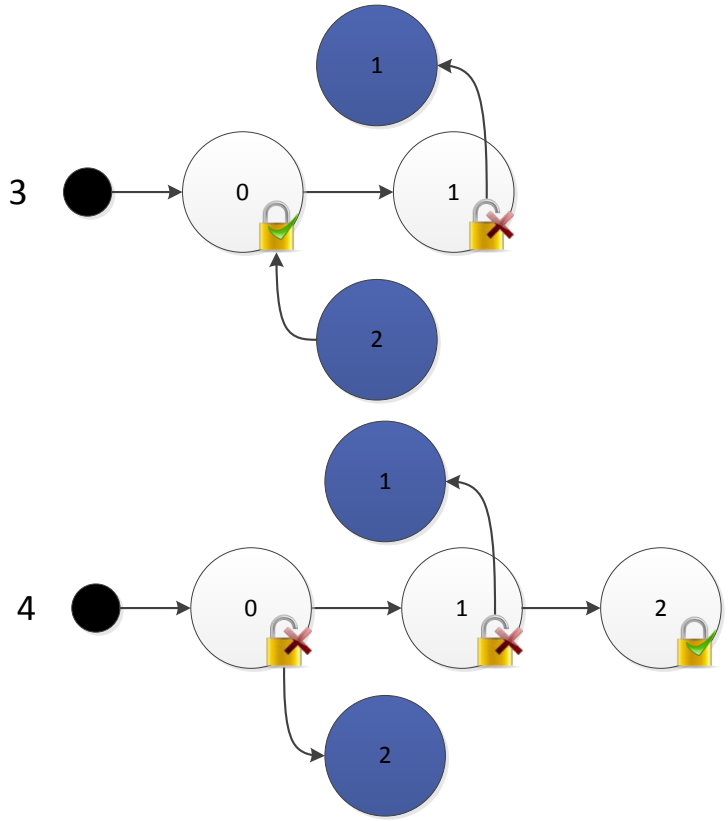


FILELISTE DURCHSUCHEN

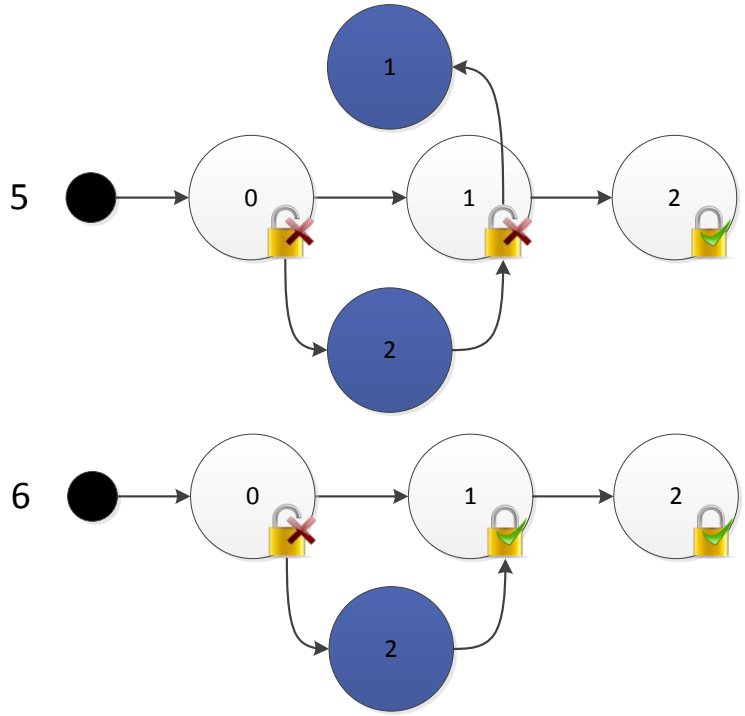
Die Liste ist so aufgebaut, dass nur in der Such und List Funktion ein Locking von den Linking Mutexen passiert. Die Such Funktion ist somit eine sehr zentrale Funktion und wird deshalb zuerst beschrieben.



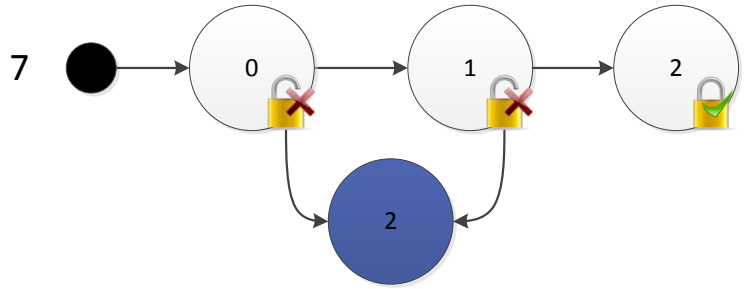
1. Thread 1 hat die Suchfunktion gestartet und hat bereits den Linking Mutex vom ersten Element gelockt und versucht den Mutex von File 1 zu locken
2. Thread 1 hat nun den Mutex von Element 0 und File 1. Thread 2 versucht den Mutex von Element 0 zu locken

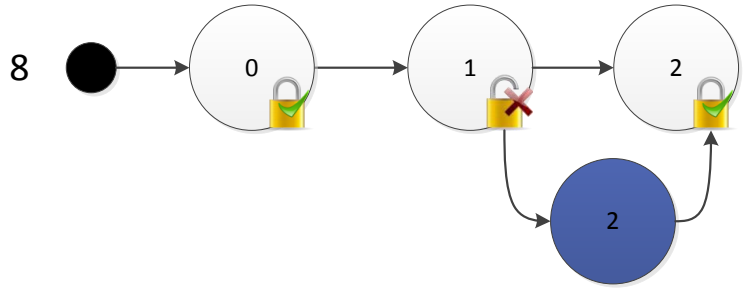


3. Thread 1 hat nach dem erfolgreichen Locking vom Mutex auf File 1 den Mutex auf Element 0 freigegeben. Thread 2 versucht immer noch den Mutex auf Element 0 zu locken
4. Thread 1 kann dank dem Lock auf File 1 das „next“ Attribut verändern und fügt einen neues File hinzu. Thread 2 hat nun den Lock auf Element 0

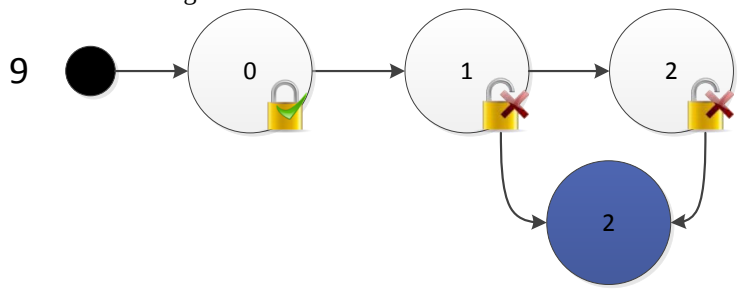


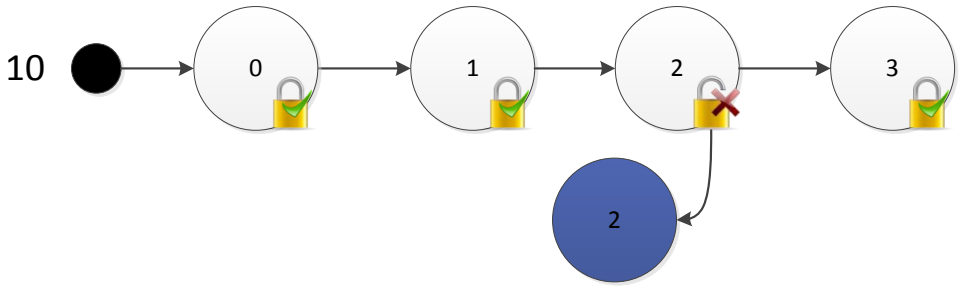
5. Thread 2 versucht nun den Mutex auf File 1 zu locken. Thread 1 ist unverändert
6. Thread 1 hat den Lock auf File 1 freigegeben und hat sich beendet. Thread 2 ist unverändert

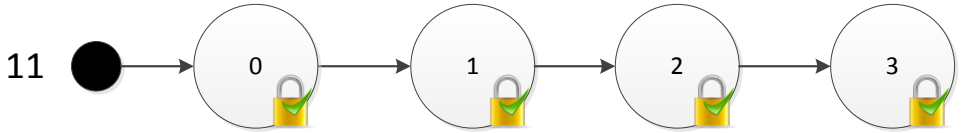




7. Thread 2 hat nun den Mutex auf Element 0 und File 1 gelockt
8. Thread 2 gibt den Mutex auf Element 0 frei und versucht den Mutex auf Element 2 zu locken



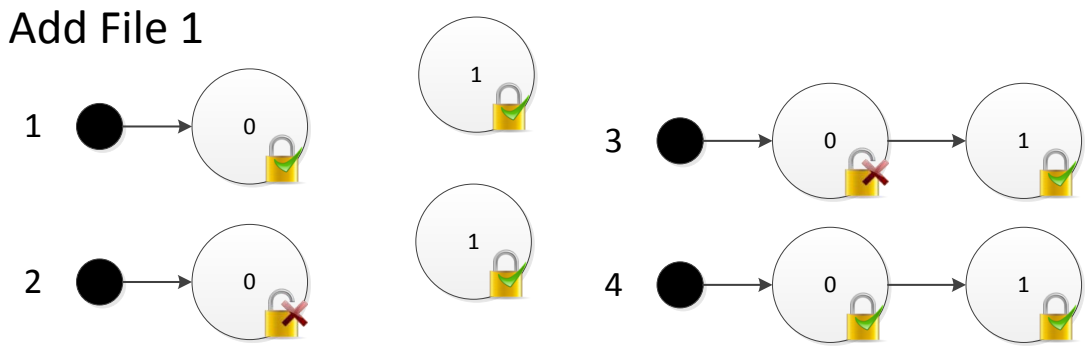




9. Thread 2 hat nun den Mutex auf File 1 und File 2
10. Thread 2 gibt den Mutex von File 1 frei und kann durch das Locking auf File 2 ein neues File hinzufügen
11. Thread 2 gibt den Mutex frei und beendet sich

FILE ERSTELLEN (CREATE)

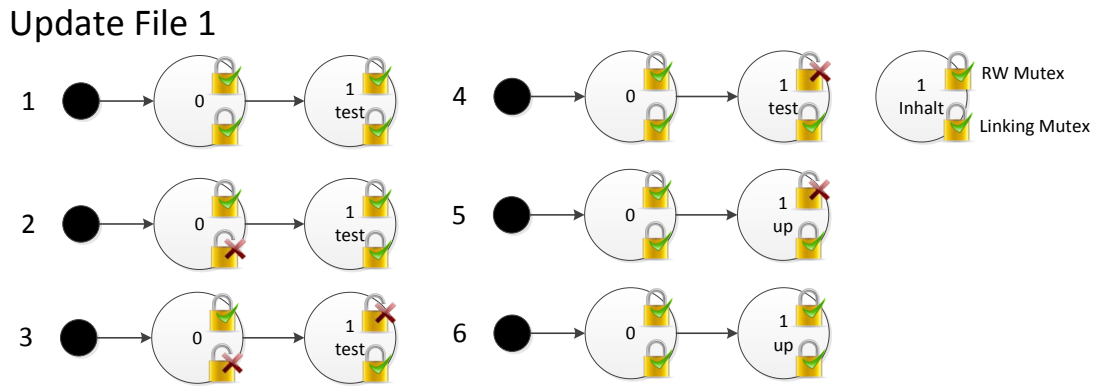
Wenn ein neues File erstellt wird, wird es an das Ende der Liste hinzugefügt, dass braucht zwar Zeit, da man jedes Mal bis zum Ende der Liste iterieren muss, jedoch muss dass sowie so gemacht werden, da der Filename nur ein Mal vorkommen darf. Das letzte Element wird von der Such Funktion ausgegeben



1. In der Liste befindet sich momentan das erste Element und ein File wird hinzugefügt
2. Der Linking Mutex vom letzten Element der Liste wird gelockt
3. Das neue File wird mit dem letzten Element verknüpft
4. Der Mutex wird freigegeben

FILE ÄNDERN (UPDATE)

Damit ein File geändert werden kann, muss es existieren und somit wird zuerst die Such Funktion ausgerufen. Die Suchfunktion lockt das File eines vor dem zu editierenden File und gibt die beiden Files dann zurück. Die Update Funktion locket dann den ReadWrite Mutex mit Write und gibt erst danach den Linking Mutex vom vorherigen File frei, damit ist sichergestellt, dass nicht eine andere Aktion das gleiche File erhält und es danach zu Inkonsistenzen führt. Das verwenden eines zusätzlichen ReadWrite Mutex hat den grossen Vorteil, dass andere Files nicht mehr davon betroffen sind, falls der Update Prozess länger dauern sollte. Es kann danach ohne Probleme über die Liste iteriert werden.



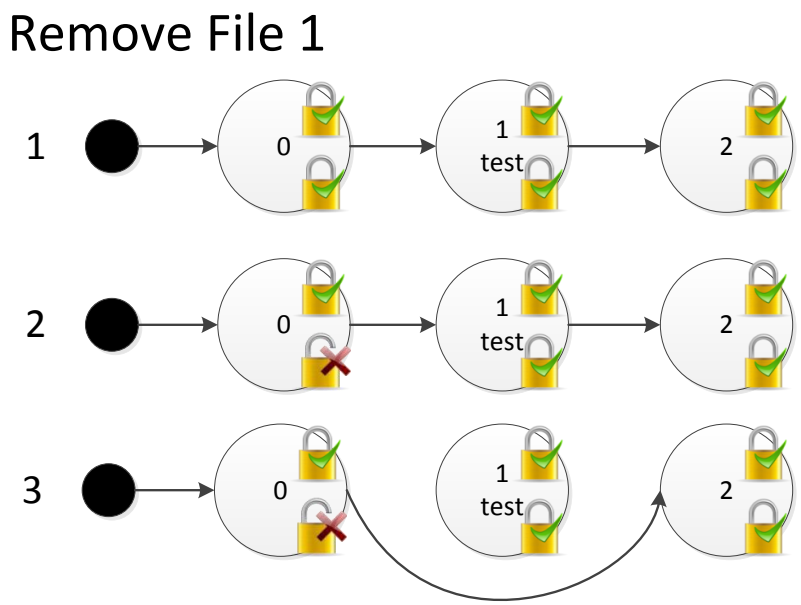
1. Start Situation
2. Der Linking Mutex vom vorherigen Element wird gelockt
3. Der ReadWrite Mutex wird mit Write gelockt
4. Der Linking Mutex vom vorherigen Element wird freigegeben
5. Der Inhalt des Files wird verändert (test → up)
6. Der Write Mutex wird freigegeben

FILE LESEN (READ)

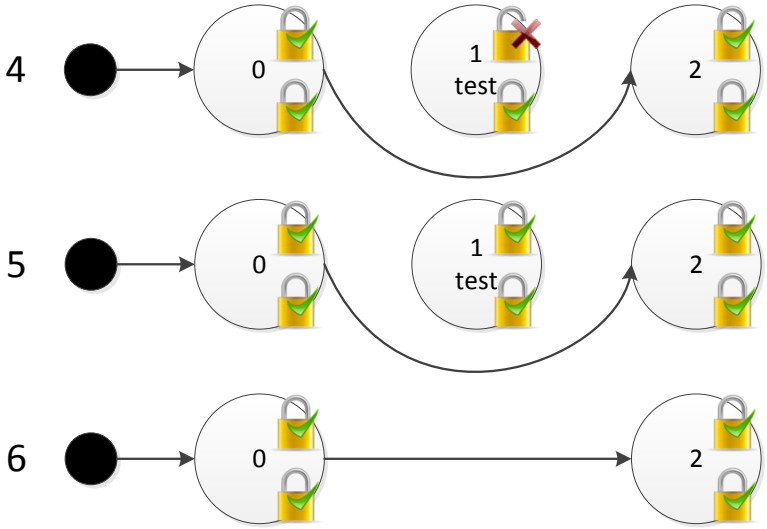
Der Lese Vorgang läuft genau gleich wie der Update Vorgang ab, mit einem kleinen Unterschied, der ReadWrite Mutex wird nur mit Read gelockt. Dies hat zur Folge, dass beliebig viele Thread gleichzeitig das File lesen können. Falls ein Update Vorgang gestartet wird, muss dieser warten, bis alle Read Prozess, welche den Mutex gelockt haben beendet sind.

FILE LÖSCHEN (DELETE)

Wie bei den vorherigen Vorgängen wird zuerst die Such Funktion gestartet, diese lockt das File eine Stelle vor dem zu löschenden File und gibt die Elemente zurück. Der Lösch Vorgang nimmt das File aus der Liste heraus und löscht es.



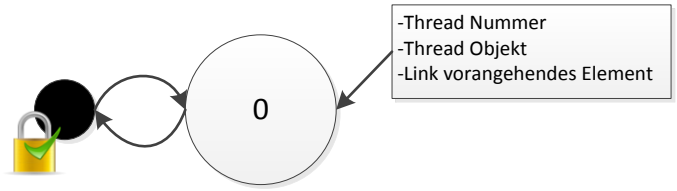
1. Start Situation
2. Der Linking Mutex vom vorherigen Element wird gelockt
3. Das File wird aus der Liste herausgenommen. Das Element 0 zeigt nun direkt auf File 2



4. Der ReadWrite Mutex wird mit Write gelockt, damit sichergestellt wird, dass keine Read oder Write Vorgänge auf diesem File am Laufen sind
5. Der ReadWrite Mutex wird freigegeben
6. Das File wird gelöscht

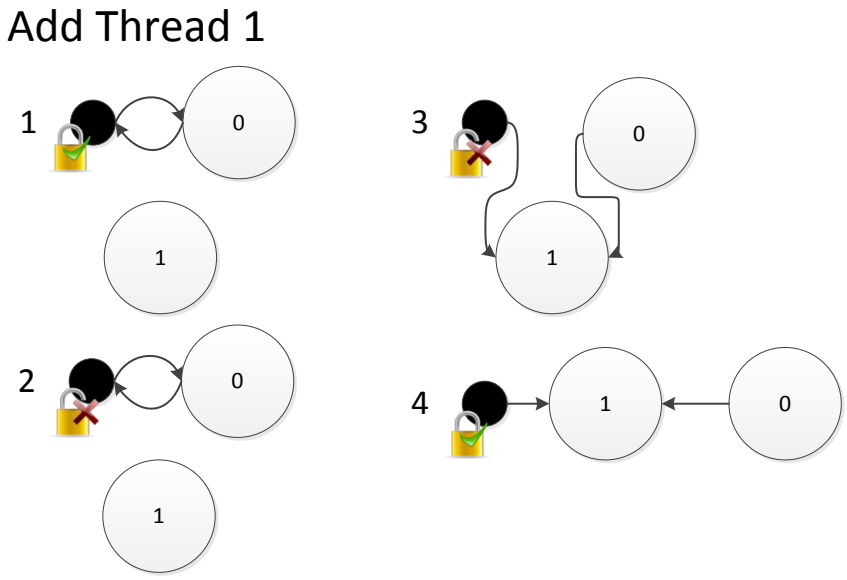
CLEANUP

Für jede Verbindung wird ein neuer Thread aufgemacht und damit diese wieder sauber aufgeräumt werden, wurde eine zusätzliche Linked List erstellt, welche für die Anwendung effizienter ist.
Der grosse Unterschied ist, dass die Elemente vorne und nicht hinten eingefügt werden. Diese Liste ist auch nur für die Add-Funktion gleichzeitig aufrufbar, die Löschfunktion bzw. der Cleanup Vorgang darf nur jeweils von einem Thread gestartet werden. Diese Liste besitzt keine Update, Read oder spezifische Delete Funktion. Weiter gibt es bei ihr nur einen Mutex und der ist für das Einfügen von neuen Elementen. Die Elemente enthalten eine Thread Nummer, das Thread Objekt an sich und einen Link zum vorangehenden Element.



THREAD HINZUFÜGEN

Die Add Funktion wird für jede Verbindung aufgerufen und sollte deshalb möglichst effizient sein, somit habe ich die Anzahl Locks und Vorgänge auf ein Minimum reduziert. Der Thread wird während dem Locking als neuer Head definiert und mit dem vorangehenden Element verknüpft.

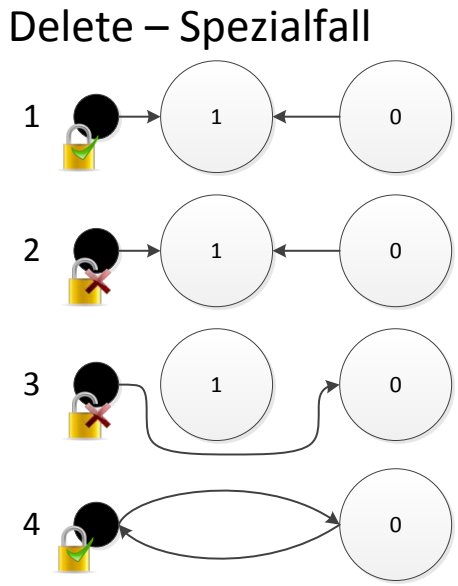


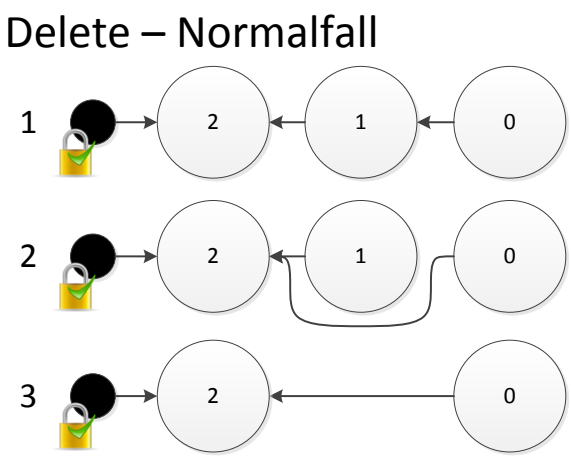
1. Thread 1 soll in die Liste hinzugefügt werden
2. Der Mutex wird gelockt, damit ist sichergestellt, dass nicht zwei Threads an den Start der Liste gehängt werden
3. Thread 1 wird als Head hinzugefügt und beim vorherigen Head als Vorangehender
4. Der Mutex wird freigegeben

CLEANUP / DELETE

Bei einem Cleanup Auruf wird vom Ende der Liste angefangen. Das Ende der Liste bildet immer das Element 0, es wird nie entfernt. Von diesem Element wird dann immer das vorangehende Element gejoined und dann aus der Liste gelöscht.

Beim Delete gibt es zwei unterschiedliche Fälle, wenn man einen Thread aus der Liste entfernt, ist kein Locking nötig, wenn man jedoch das letzte Element aus der Liste entfernt, muss man den Lock beachten.





Spezialfall

1. Thread 1 soll gelöscht werden, da Element 0 immer in der Liste bleibt, ist es das letzte Element in der Liste
2. Der Mutex wird gelockt
3. Der Head wird auf das Element 0 gesetzt und das „prev“ Attribut vom Element 0 wird auf NULL gesetzt
4. Der Mutex wird freigegeben

Normalfall

1. Thread 1 soll gelöscht werden
2. Das „prev“ Attribut von Element 0 wird auf Thread 2 gesetzt
3. Thread 1 wird gelöscht

OPTIMIERUNG

Für die Optimierung der Applikation wurde valgrind (Valgrind) verwendet. Das Tool zeigt Memory Leaks auf und ist sehr mächtig. Es benötigt etwas Einarbeitungszeit, aber danach ist es sehr hilfreich. Es wurden alle Tests und Test-Clients durchgeführt und mit valgrind überwacht, um sicherzustellen, dass keine Memory Leaks im Server vorhanden sind. Durch valgrind wurde erkannt, dass beim Erstellen der Files aus Performance-Gründen das File erstellt wird, bevor die Suche beginnt, da sonst die Locking-Zeit etwas länger wäre, wenn der Filename jedoch schon vorhanden ist, wurde der Speicherplatz des Files nicht freigeben sondern die Methode wurde einfach beendet. Es hat sich auch herausgestellt, dass die Funktion für die String Verkettung sehr Memory Leak anfällig war und diese wurde im Logging Bereich sehr häufig verwendet. Das führte dazu, dass der Server bei einem hohen Debug-Level oft abstürzte. Valgrind hat ebenfalls herausgefunden, dass ein Unlock von einem Mutex gemacht wurde, obwohl es nicht nötig gewesen wäre. Es wurden alle gefunden Leaks und Mängel beseitigt und der Server lief danach viel flüssiger.

RÜCKBLICK

Die Aufgabe war für mich sehr schwierig zu meistern und hat viel mehr Zeit gekostet als am Anfang gedacht. Am Anfang hatte ich zuerst ein wenig Probleme mit dem ganzen Aufbau, danach stand der erste Entwurf relativ schnell einmal. Dann kam aber die mühsame Arbeit der Optimierung, dass der Server auch stabil läuft. Ich hatte viele Segmentation Faults und wusste nicht wo sie auftraten. Schlussendlich habe ich an vielen Stellen noch nachbessern müssen. Es stellt sich zum Beispiel heraus, dass viele SegV von dem Logger kamen, da die Message länge zu lang war, ich habe danach vsnprintf anstatt vsprintf verwendet um sicher zu gehen, dass dies nicht mehr passiert.

Die Arbeit war sehr spannend und ich hab jetzt ein viel besseres Verständnis für C. Ich kenne nun einige Patterns und komme auch mit dem Programm make besser zu Gange. Es war sehr toll die Limits des Servers auszureizen und ihn noch stabiler und schneller zu machen. Wirklich mühsam an C sind die Segmentation Faults und Core Dumps, an welche ich mich immer noch nicht gewöhnt habe, vor allem in Kombination mit Concurrency. Es gab einige Stunden, in denen ich versuchte den Ablauf zu reproduzieren und den Fehler zu finden und fast verzweifelt bin. Ich denke, da hab ich noch sehr viel Potenzial mein Verständnis zu erweitern.

Es war auch spannend, dass ich den Server mit einem Java Client testen konnte. Das hat mir am Anfang sehr geholfen, da ich mich in dieser Welt einfach schon auskannte und so mit JUnit Test schreiben konnte. CUnit, welches ich danach noch für Unit Tests in C verwendet habe, ist ok, aber bei weitem natürlich nicht so mächtig wie JUnit. Meines Wissens ist es zum Beispiel nicht möglich eine Erwartet/Aktuell-Ausgabe bei einem fehlgeschlagenen ASSERT ausgeben zu können, was eine sehr wichtige Funktion ist. Ich bin aber sehr erstaunt, wie viele Helfer-Tools es für C gibt, mit CUnit und mit valgrind hat man zwei sehr hilfreiche Tools, in welchen man relativ schnell fit ist.

Da ich die Funktionen in separate Files unterteilt habe, habe ich nun eine eigenen kleine Library von Helfer in C (wie zum Beispiel den Logger, die Concurrent Linked List oder Thread List), welche für weiter Projekte sehr nützlich werden können. Alles in allem eine sehr spannende und herausfordernde Aufgabe, welche mein Verständnis für C und Concurrency verbessert hat und wie vorher erwähnt wiederverwendbare Elemente zum Ergebnis hatte.

ANHANG

LITERATURVERZEICHNIS

POSIX Bracket Expressions. (kein Datum). Abgerufen am 30. April 2014 von http://www.regular-expressions.info/posixbrackets.html

Using regular expressions in C. (kein Datum). Abgerufen am 14. April 2014 von http://www.lemoda.net/c/unix-regex/

Valgrind. (kein Datum). Abgerufen am 19. Juni 2014 von http://valgrind.org/