CONCURRENT PROGRAMMING IN C

FileServer

Zürcher Hochschule für angewandte Wissenschaften
Seminar: Concurrent Programming in C
Dozent: Nico Schottelius
Erscheinungsjahr: 2014

Eine Arbeit von Severin Andrew Müller

Inhaltverzeichnis

1. Themenwahl und Aufgabenstellung

1.1 Eigenmotivation

Als ich 2002 mit der Programmierung anfangen wollte, machte ich mir Gedanken darüber, welche Programmiersprache ich als erstes erlernen möchte. Da mein Vater für Studer Revox die Software für die elektronischen Geräte in C schrieb, entschied ich mich für eben diese Sprache, damit ich auch mal nach einem Tipp fragen konnte.

2008 begann ich mit einem Projekt, das meine Kenntnisse in C vertiefen sollte. Dieses sah einen Zusatzserver zu einem IRC Server vor. Dazu musste ich mir Kenntnisse in der Programmierung von Sockets und Datenbanken aneignen.

In dieser Arbeit habe ich mich für das Thema „Fileserver“ entschieden, weil ich so die Konzepte, die ich in meinem früheren Projekt erarbeitet habe noch vertiefter erlernen konnte.

1.2 Abgrenzung der Aufgabenstellung

In der Aufgabenstellung wurde ein Fileserver verlangt, der Concurrency fähig ist. Das heisst, dass mehrere Clients gleichzeitig darauf zugreifen können müssen, ohne dass diese sich in die Quere kommen.

Folgende Randbedingungen galt es einzuhalten:

- keine Global Locks
- Das physische Dateisystem darf nicht benutzt werden
- Pthreads oder fork und shm
- Kommunikation über TCP/IP oder Unix Domain Sockets
- Indeces beginnen bei 0
- Client muss nie retry machen
- Das Protokoll muss eingehalten werden

Ich habe mich für eine Pthreads Lösung mit TCP/IP entschieden und möchte die Realisierung auf den folgenden Seiten darlegen.

1.3 Anleitung zur Nutzung

1.3.1 Aufsetzen

Um die Applikation zu nutzen muss das Programm mit make erstellt werden. Zunächst muss das Repository gezogen werden:

```
user@linux:~$git clone https://github.com/fish-guts/concurrent.git
```

Danach können wir den Testclient erstellen. Dazu müssen wir uns im Rootverzeichnis des Projektes befinden.

```
user@linux:~/concurrent$ make test
```

Zu guter Letzt erstellen wir den Server:

```
user@linux:~/concurrent$ make run
```

1.3.2 Betrieb

Sobald wir die beiden binaries erstellt haben können wir den Server starten.

```
user@linux:~/concurrent$ ./run
```

```
################################

Welcome to Severin's FileServer

################################



Starting server...Bind Successful

Server started successfully, listening on port 8083
```

Wenn wir diesen Screen sehen, läuft der Server und wartet auf Verbindungen.

Wir können nun einen Client starten der auf den Server verbindet. Dazu öffnen wir ein zweites Terminal Fenster und geben folgendes ein.

```
user@linux:~/concurrent$ ./test
```

Konnte der Client erfolgreich verbinden ist folgendes zu sehen:

```
Connected to server
```

Nun können wir im Client Befehle an den Server schicken. Gültige Befehle sind:

- create <Filename> <Filesize>\n <Content>
- update <Filename> <Filesize>\n <Content>
- delete <Filename>
- read <Filename>
- list
- exit

2. Realisierung

2.1 Lösungsansatz

Wie bereits in Kapitel 1.2 erwähnt habe ich mich in dieser Arbeit für eine Pthreads Lösung entschieden.

Der Vorteil, den ich hier sehe ist die Tatsache, dass Threads untereinander die gleichen Ressourcen teilen[1], so also auf Shared Memory verzichtet werden kann.

Um einen Lock zu erzeugen werden Mutexe eingesetzt. Ein Mutex ist im Wesentlichen ein Lock den wir setzen, bevor auf eine Ressource zugreifen, die auch von anderen verwendet werden darf[2]. Wenn mehrere Threads gleichzeitig auf eine Ressource zugreifen möchten so setzt der erste Thread den Mutex und die anderen Threads werden in einen wartenden Zustand versetzt bis der Lock wieder gelöst wird[3].

Da eines der Kriterien war, dass keine Global Locks verwendet werden dürfen, musste ich mir Gedanken darüber machen, wie ich den Lock auf Dateiebene setze.

Um dieses Problem zu lösen habe ich mich für eine globale Liste mit den Dateien entschieden die in Form einer verketteten Liste vorliegt, damit der Lock auf einfach Weise auf die Datei gesetzt werden kann.

```
struct _sFile {
    char *filename;
    size_t size;
    char *content;
    sFile *next;
    pthread_mutex_t mutex;
};
```

Abb. 1 Dateistruktur

Jeder Knoten dieser (einfach) verketteten Liste stellt eine virtuelle Datei dar.

Erstellen wir nun eine neue Datei, wird diese als neuer Knoten an der Liste angehängt:

Abb. 2: Verzeichnisstruktur

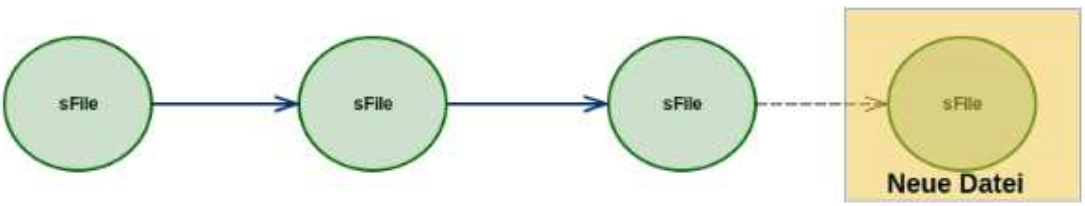

Abb. 3: Neue Datei anlegen

Dies eröffnet uns nun die Möglichkeit, einen Mutex auf den jeweiligen Knoten zu setzen.

Was passiert aber, wenn auf dem nachfolgenden Knoten, der nicht gesperrt ist, gleichzeitig eine Datei angelegt wird? Das würde dazu führen, dass das Programm abstürzt, weil beide neue Dateien in den gleichen Speicherbereich schreiben wollen.

Dasselbe Problem tritt auf, wenn man eine Datei löschen will und gleichzeitig die nachfolgende Datei gelöscht wird. Da beim Löschen eines Knotens in der Liste der entsprechende Speicherbereich freigegeben wird und der Zeiger dafür auf den übernächsten Knoten zeigen soll, zeigt dieser ins Nirvana wenn der nachfolgende Knoten gleichzeitig gelöscht wird.

Die Lösung: Wir verwenden einen doppelten Lock. Die Idee dabei ist, dass wenn wir durch die Liste durchgehen wollen, um eine Datei zu löschen, oder eine Datei anlegen wollen, wir jeweils auch den nächsten Knoten locken, damit der Zeiger vom Knoten der bearbeitet wird weiterhin auf einen gültigen nächsten Knoten zeigt.

In Abb. 4 sehen wir das Prinzip dieses doppelten Locks. Damit der Zeiger vom zweiten File auf das vierte File zeigen kann, wenn das dritte File gelöscht wird, müssen wir dieses auch locken. Wenn wir durch die Liste iterieren, verschieben wir jeweils auch die Locks paarweise. Mit diesem Ansatz stellen wir sicher, dass die Liste integer bleibt.

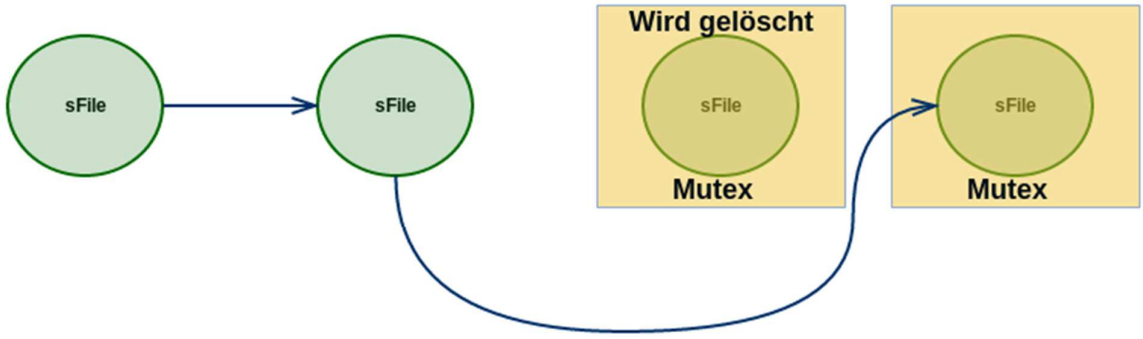

Abb. 4: Löschen einer Datei

2.2 Implementierung Server

2.2.1 Einleitung

Wie einleitend erwähnt habe ich bereits Erfahrungen mit Servern sammeln können. Diese wollte ich mir hier zunutze machen indem ich ähnliche Konzepte zu diesem Zweck verwendet habe.

2.2.2 Verbindung

Beim Serverstart öffnet das Programm einen Socket und geht dann in eine Endlosschlaufe über um die Verbindung aufrecht zu erhalten. Zudem erstellen wir eine Dummydatei und legen diese in unsere globale Dateiliste. Diese soll als Listenheader agieren und wird bei einem LIST Befehl nicht angezeigt.

```
while (!quitting) {
    len = sizeof(struct sockaddr_in);

    int client_sock = accept(sock, (struct sockaddr *) &client, &len);
    if (client_sock < 0) {
        fprintf(stderr, "Accept failed\n");
    } else {
        /* This is the client process */
        int *sock_ptr = (int *) malloc(sizeof(int));
        *sock_ptr = client_sock;
        tid = pthread_create(&threadlist[thread_count], NULL, doprocessing, sock_ptr
        pthread_detach(threadlist[thread_count]);
        thread_count++;
        if (tid) {
            fprintf(stderr, "Error creating thread: %d\n", tid);
        }
    }
```

Abb. 5: Code Serververbindung

In dieser Schleife wartet der Server nun auf Verbindungen. Verbindet sich ein Client zum Server wird für diesen Client an neuer Thread erzeugt und die Funktion doprocessing() wird aufgerufen, welche wiederum in einer Endlosschlaufe für diesen Thread läuft. Jeder Client erhält einen eigenen Socket und wir übergeben doprocessing() einen Zeiger auf diesen Socket.

2.2.2 Server Funktionen allgemein

Unser Server soll bekanntlich in der Lage sein Befehle auszuführen. Ich wollte, dass die Funktionen dynamisch und nicht über if-then-else Vergleiche aufgerufen werden konnten. Um dies zu bewerkstelligen habe ich eine Struktur angelegt, die einen String für den effektiven Namen und einen Zeiger auf eine Funktion beinhaltet:

```
struct _cmd
{
    const char *name;
    void (*func)(int s,int ac, char **av);
};
```

Abb. 6: Funktionsstruktur

Nun können wir ein Array erzeugen, die alle benötigten Funktionen beinhalten und wir diese über den String aufrufen können:

```
cmd cmds[] = {
        { "CREATE", cmd_create },
        { "EXIT" ,  cmd_exit },
        { "LIST", cmd_list },
        { "READ", cmd_read },
        { "DELETE", cmd_delete },
        { "UPDATE", cmd_update },
        { NULL, NULL }
};
```

Abb. 7: Liste der Funktionen

Wenn der Client nun einen Befehl an den Server schickt, durchsuchen wir dieses Array nach einem

Befehl der passt:

```
cmd *find_cmd(const char *name) {
    cmd *c;
    for (c = cmds; c->name; c++) {
        if (stricmp(name, c->name) == 0) {
            return c;
        }
    }
    return NULL;
}
```

Abb. 8: Suchen eines Serverbefehls

Wird kein entsprechender Befehl gefunden, schickt der Server eine entsprechende Meldung an den Client.

Wie kommt aber der Befehl vom Client an dieses Array? Um eine möglichst dynamische Gestaltung der ganzen Befehle zu erreichen habe ich eine Prozessormethode eingebaut die folgendes tut:

- Entgegennahme des Befehls
- Aufsplittung des Befehls in Befehl und Argumente
- Ablegen der Argumente in einem Array
- Zählen der Argument (macht das überprüfen der korrekten Anzahl Argumente einfacher)
- Weiterleitung an die Funktion find_cmd()

Die genaue Implementierung ist der Funktion **process()** in der Datei main.c und **tokenize()** in der Datei command.c nachzulesen.

2.2.3 Iterieren von Mutexen

Wie in Kapitel 2.1 beschrieben setzen wir für jede Operation auf dem Dateisystem zwei Mutexe die wir dann jeweils miteinander verschieben wenn wir durch die Liste iterieren müssen. Dafür habe ich frei Funktionen implementiert; eine die den Iterator initialisiert, eine die jeweils beide Mutexe zum nächsten Knoten schiebt, und eine, die den Iterator auflöst

```
void iterator_init(iterator *it) {
    it->a = NULL;
    it->b = file_list;
    pthread_mutex_lock(&it->b->mutex);
}
```

Abb. 9: Funktion zum Initialisieren des Iterators

```
void iterator_destroy(iterator *it) {
    if (it->a != NULL)
        pthread_mutex_unlock(&it->a->mutex);
    if (it->b != NULL)
        pthread_mutex_unlock(&it->b->mutex);
}
```

Abb. 10: Funktion zum Aufheben des Iterators

```
sFile *iterator_next(iterator *it) {
    if (it->a != NULL)
        pthread_mutex_unlock(&it->a->mutex);
    if (it->b->next == NULL) {
        pthread_mutex_unlock(&it->b->mutex);
        return NULL;
    }
    it->a = it->b;
    it->b = it->b->next;
    pthread_mutex_lock(&it->b->mutex);
    return it->b;
}
```

Abb. 11: Funktion für den Iteratorsuchlauf

2.2.4 Datei erstellen (CREATE)

Wenn wir eine Datei erstellen wollen, senden wir den Befehl CREATE <filename> <filesize>an den Server. Dieser wartet dann auf eine zweite Nachricht welche den Inhalt enthält.

Wir durchsuchen zunächst die Liste um festzustellen, ob eine solche Datei bereits existiert:

```
if (it.b->next!=NULL)
    while ((current = iterator_next(&it)) != NULL) {
        if (stricmp(current->filename, av[1]) == 0) {
            // file found, send error and abort
            iterator_destroy(&it);
            free(buf);
            send(s, ext, sizeof(ext), 0);
            return;
        }
        if (it.b->next==NULL)
            break;
    }
```

Abb. 12: Erstellen einer Datei 1

```
        // file not found, create new one
        sFile *f = scalloc(sizeof(sFile), 1);
        f->filename = sstrdup(av[1]);
        f->size = content_size;
        f->content = sstrdup(buf);
        f->next = NULL;
        pthread_mutex_init(&f->mutex,NULL);
        it.b->next=f;
        pthread_mutex_destroy(&f->mutex);
        iterator_destroy(&it);
        //done, send response
        send(s, suc, sizeof(suc), 0);
}
```

Abb. 13: Erstellen einer Datei 2

Ist dies der Fall wird eine entsprechende Meldung an Client gesendet. Ansonsten kann der Client nun eine zweite Meldung mit dem Inhalt schicken und die Datei wird angelegt:

2.2.5 Datei bearbeiten (UPDATE)

Möchte ein Client eine Datei updaten, so kann er den Befehl UPDATE <Filename><Filesize> senden. Existiert diese Datei wartet der Server auf den Inhalt den der Client in einer zweiten Nachricht dann senden kann.
Die Implementierung sieht im Wesentlichen gleich aus wie bei CREATE, der Unterschied besteht darin, dass wenn das File bei der Suche gefunden wird, die Informationen entsprechend überschrieben werden.

2.2.6 Datei lesen (READ)

Der Client den Inhalt einer Datei lesen möchte, sendet er den Befehl READ <Filename>.

Der Server gibt dann den Inhalt der Datei zurück.

```
void cmd_read(int s, int ac, char **av) {
    char err[] = "NOSUCHFILE\n";
    char *filename = strdup(av[1]);
    char *fileInfo = (char*) malloc((sizeof(char*) * 1024));
    char *fileContent = (char*) malloc((sizeof(char*) * 4096));
    iterator it;

    iterator_init(&it);
    sFile *current;
    while ((current = iterator_next(&it)) != NULL) {
        if ((stricmp(filename, current->filename) == 0)) {
            sprintf(fileInfo, "READ %s\n", current->filename);
            sprintf(fileContent, "%s\n", current->content);
            send(s, fileInfo, (int) strlen(fileInfo), 0);
            send(s, fileContent, (int) strlen(fileContent), 0);
            iterator_destroy(&it);
            return;
        }
    }
    send(s, err, (int) sizeof(err), 0);
    iterator_destroy(&it);
}
```

Abb. 14: Lesen einer Datei

Existiert das File nicht, so erhält der Client eine entsprechende Meldung.

2.2.7 Datei löschen (DELETE)

Mit DELETE <Filename> kann der Client eine Datei löschen. Dabei wird einfach der Zeiger auf die Datei auf den übernächsten Knoten „umgehängt“ (siehe auch Abb. 4).

```
while ((current = iterator_next(&it)) != NULL) {
    if ((stricmp(filename, current->filename) == 0)) {
        pthread_mutex_init(&current->mutex,NULL);
        it.a->next = it.b->next;
        pthread_mutex_destroy(&current->mutex);
        iterator_destroy(&it);
        send(s, suc, sizeof(suc), 0);
        return;
```

Abb. 15: Löschen einer Datei

2.2.8 Dateien anzeigen (LIST)

Um alle Dateien anzuzeigen sendet der Client den Befehl LIST. Der Server sendet dann eine Liste

mit allen Dateinamen an den Client.

Hier besteht jedoch die Problematik, dass in der Zwischenzeit eine Datei gelöscht werden könnte und globale Locks sind in dieser Arbeit nicht erlaubt.

Die Lösung: Ich schreibe die Resultate in einen Puffer und gebe am Ende diesen aus.

```
sFile *current = file_list;
iterator_init(&it);
// write files to buffer to make it threadsafe
while ((current = iterator_next(&it)) != NULL) {
    sprintf(buf,"%s%s\n",buf,current->filename);
    count++;
}
```

Abb. 16: Auflistung Dateien mit Puffer

2.2 Implementierung Client

Der Fokus in dieser Arbeit liegt auf dem Serverteil. Der Client ist zum Testen gedacht und ist daher relativ simpel gestaltet.

Im Wesentlichen wird eine Verbindung zum Server erstellt und zwei Threads erzeugt: einer der in den Socket schreibt, einer der vom Socket liest. Der Grund für diese Implementierung ist die Tatsache, dass einige Befehle zwei Nachrichten erfordern. Die Implementierung des Clients kann man in der Datei client.c nachlesen.

2.3 Sonstiges

In meinen bisherigen C-Projekten habe ich jeweils einen Satz Hilfsfunktionen geschrieben. Auf diese wollte ich auch in diesem Projekt nicht verzichten, daher sind diese auch eingebaut worden. Die Funktionen können in der Datei util.c nachgelesen werden.

3. Probleme und Lösungen

Bei der Erstellung dieser Arbeit trat doch das eine oder andere Problem auf. Mit dem Thema Concurrency hatte ich mich vorher noch nie auseinandergesetzt. Dementsprechend war die Konzeptionierung eher Zeitaufwändig. Das Studium der empfohlenen Lektüre in Verbindung mit der Vorlesung „Systemsoftware“ schuf hier Abhilfe und hatte sich sehr gelohnt und ich habe mich sehr gefreut, etwas darüber zu lernen.

Ein weiteres Hindernis waren die fehlenden Kenntnisse über die Erstellung von Make-Files. Das Thema ist umfangreich und erforderte einen gewissen Aufwand.

Wie bei der Erstellung von solchen Programmen nicht unüblich hatte auch ich mit Segemtierungsfehlern zu kämpfen. Durch meine Erfahrung in der C-Programmierung war es mir jedoch möglich diese Fehler soweit wie möglich auszumerzen. Mit dem Debugger in Eclipse und Debug Ausgaben konnte diese Problem jeweils „einkesseln“ und so den Fehler lokalisieren und beheben.

4. Fazit

4.1 Rückblick

Die Bearbeitung dieses Seminars erwies als äusserst zeitaufwändig, weil fundierte Kenntnisse in einigen Komponenten notwendig waren. Dennoch war die Aufgabenstellung sehr interessant, weil ich mich wie erwähnt bereits in der Vergangenheit mit der Programmierung von Sockets beschäftigt hatte. Am schwierigsten war der Entwurf der Lösung der doch einiges an Lesearbeit verursachte und dadurch der zeitliche Rahmen dieser Arbeit gesprengt wurde.

4.2 Schlussfolgerungen

Pthreads eigenen sich sehr gut um eine Applikation zu schreiben, die „concurrent“ Operationen ermöglichen soll. Das Konzept der Mutexe ermöglicht es, Threads warten zu lassen bis der Lock wieder freigegeben wurde. Ich konnte ein weiteres Konzept in der Programmiersprache C erlernen was mir sicherlich nützlich sein kann.

Frappant finde ich die Unterschiede in der Komplexität im Vergleich zur Programmiersprache Java; das Arbeiten mit Threads ist in Java einiges einfacher, was an den Grundsätzen der jeweiligen Sprachen liegt. In Java brauche ich mich nicht über irgendwelche Speicherallozierungen zu kümmern und das synchronisieren von Threads ist einfacher zu bewerkstelligen. Dennoch haben Pthreads und C ihre Daseinsberechtigung – in performancekritischen Applikationen wird C immer wieder zum Einsatz kommen.

4.3 Dank

Ich möchte mich bei allen bedanken die mich bei der Erstellung dieser Arbeit in irgendeiner Form untersützt haben.

5. Anhang

Anhang A: Bilderverzeichnis

Anhang B: Literaturverzeichnis

[1]: Stevens, W. Richard: Advanced Programming in the UNIX Environment. Hallbergmoos, Deutschland: Pearson Deutschland, 3. Auflage, 2013, Seite 383, ISBN 978-0-321-63773-4
[2]: Stevens, W. Richard: Advanced Programming in the UNIX Environment. Hallbergmoos, Deutschland: Pearson Deutschland, 3. Auflage, 2013, Seite 399, ISBN 978-0-321-63773-4
[3]: Stevens, W. Richard: Advanced Programming in the UNIX Environment. Hallbergmoos, Deutschland: Pearson Deutschland, 3. Auflage, 2013, Seite 400, ISBN 978-0-321-63773-4

Sonstige Quellen:

Kernighan, Brian; Ritchie, Dennis: The C Programming Language, London, Vereinigtes Königreich, Prentice Hall, 50. Auflage, 2012
ISBN 0-13-110362-8

http://en.wikipedia.org/wiki/POSIX_Threads, abgerufen im Mai und Juni 2014