Multiuser-Fileserver in C

Seminararbeit Concurrent Programming in C

Niki Hausammann, i10b, hausanik@students.zhaw.ch
Dozent: Nico Schottelius

22. Juli 2014

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Abgabebedingunen . . . 3
- 1.2 Aufgabenstellung . . . 3

2 Anleitung **5**
- 2.1 Vorbereitung . . . 5
- 2.2 Starten der Server-Applikation . . . 5
- 2.3 Starten der Client-Applikation . . . 5
- 2.4 Datei erstellen . . . 5
- 2.5 Datei lesen . . . 6
- 2.6 Auflisten von Dateien . . . 6
- 2.7 Datei ändern . . . 6
- 2.8 Datei löschen . . . 7

3 Planung / Vorbereitung **8**
- 3.1 Variante 1: Shared Memory pro Datei und Double Linked List für Kontrollstruktur . . . 8
- 3.2 Variante 2: Shared Memory pro Datei und Array als Kontrollstruktur . . . 8
- 3.3 Variante 3: Nur eine Struktur . . . 9
- 3.4 Variantenentscheid . . . 9
- 3.5 Thread oder Fork . . . 9
- 3.6 Kontrollstruktur . . . 9
- 3.7 Speichern der Datei . . . 9
- 3.8 Locking-Mechanismus . . . 9
- 3.9 Kommunikation . . . 10

4 Probleme / Wege / Lösungen **11**
- 4.1 Shared Memory pro Datei . . . 11
- 4.2 Semaphore - zu viele . . . 11

5 Fazit **13**

Anhang **13**

1 Einleitung

Im Rahmen des Moduls Betriebssysteme im Studiengang Informatik an der Zürcher Hochschule für angewandte Wissenschaften muss das Seminar Concurrent Programming in C absolviert werden. Hauptbestandteil dieser Arbeit ist das erstellen einer Serveranwendung, welche Fragen eines Clients über Netzwerkverbindungen beantwortet. Die Aufgabenstellung und Rahmenbedinungen sind allesamt vom Betreuer Nico Schottelius vorgegeben. Die vollständigen Dokumente sind im Github-Repository von Nico Schottelius zu finden. Es standen zwei Aufgabenstellungen zur Verfügung, wobei für diese Arbeit das Projekt "Multi-User Fileserver" gewählt wurde.

1.1 Abgabebedingunen

Folgende Rahmenbedingunen wurden zu Beginn des Seminars definiert

- Die Abgabe der Arbeit hat über ein vorher kommuniziertes Github-Repository zu erfolgen. Der Dozent zieht den Stand des Repositories zum Abgabezeitpunkt zur Bewertung bei.
- Die Applikation muss unter Linux lauffähig sein.
- Mittels einem Makefile soll die Applikation kompiliert werden können.
- Im Verzeichnis "run" muss die ausführbare Serverapplikation liegen. Diese sollte nicht abstürzen und keine SEGV auslösen.
- Im Verzeichnis "test" soll die Testapplikation abgelegt werden.
- Eine Datei "doc.pdf" muss existieren, in welcher sich die Dokumentation der Seminararbeit befindet.

1.2 Aufgabenstellung

Die vollständige Aufgabenstellung ist auf den bereits erwähnten Github-Repository zu finden. Einige der wichtigsten Informationen sind hier noch einmal aufgeführt:

- Keine globalen Locks
- Kommunikation über TCP/IP
- Für jede Verbindung muss ein eigener Prozess oder Thread erstellt werden (fork empfohlen)

- Schwergewichtig wird der Server-Teil betrachtet
- Das Dateisystem darf nicht verwendet werden
- Locking auf Dateiebene

2 Anleitung

2.1 Vorbereitung

Wenn das Gibthub-Repository lokal vorhanden ist, so kann mit dem Befehl "make“ begonnen werden. Die Server- und Client-Applikation werden dann kompiliert, so dass im Verzeichniss "run“ die Serverapplikation und im Verzeichniss "test“ die Clientapplikation zur Ausführung bereit liegen.

2.2 Starten der Server-Applikation

Die zuvor mit "make“ erstellte Applikation kann wie folgt aufgerufen werden:

```
./server.o <lokaler TCP/IP-Port>
```

Beachtet muss werden, dass die Verwendung von bestimmten Ports lokale Administrationsrechte benötigt. Um dies zu umgehen sollte eine Port-Nummer grösser als 1023 verwendet werden.

2.3 Starten der Client-Applikation

Um mit dem Server zu kommunizieren wird die Client-Applikation verwendet. Diese kann wie nachfolgend beschrieben gestartet werden:

```
./client.o <IP-Adresse Server> <TCP/IP-Portnummer Server>
```

Um den Client erfolgreich zu starten muss der Server bereits gestartet sein, da sonst keine erfolgreiche Verbindung hergestellt werden kann.

2.4 Datei erstellen

Um eine Datei zu erstellen wird der Befehl "CREATE“ mit folgender Syntax verwendet:

```
Befehl: CREATE Dateiname Laenge
Beispiel:
Eingabe Client: REATE testfile 6
Server-Antwort: CONTENT oder FILEEXISTS
Eingabe Client: Inhalt
Server-Antwort: FILECREATED
```

Der Befehl wird mit der Eingabetaste bestätigt und zum Server gesendet, bei einer erfolgreichen Übertragung kommt die Antwort "CONTENT", welche bedeudet, dass der Server nun bereit ist den Inhalt der Datei zu empfangen. Dieser kann wiederum in der Konsole eingegeben und mit der Eingabetaste bestätigt werden. Falls die Datei erfolgreich erstellt wurde erscheint die Rückmeldung "FILECREATED". Nun kann der nächste Befehl eingegeben werden.
Falls die Datei bereits existiert, kommt nach der ersten Eingabe bereits die Rückmeldung "FILEEXISTS".

2.5 Datei lesen

Der Inhalt einer Datei wird mit dem Befehl "READ" gelesen. Der Befehl muss wie folgt eingegeben werden:

```
Befehl: READ Dateiname
Beispiel:
Eingabe Client: READ testfile
Server-Antwort: FILECONTENT testfile 6 oder NOSUCHFILE
Server-Antwort: Inhalt
```

In der Antwort ist der Dateiname, die Länge und danach noch der Inhalt enthalten.

2.6 Auflisten von Dateien

Das Dateiverzeichnis kann mit "LIST" angezeigt werden. Dieser Befehl hat keine weiteren Parameter und listet alle Dateien auf.

```
Befehl: LIST
Beispiel:
Eingabe Client: LIST
Server-Antwort: ACK 1
Server-Antwort: testfile
```

2.7 Datei ändern

Für Änderungen an einer bestehenden Datei kann der Befehl "UPDATE" verwendet werden. Damit kann der Inhalt einer bereits bestehenden Datei überschrieben werden.

```
Befehl: UPDATE Dateiname Laenge
Beispiel:
Eingabe Client: Update testfile 14
Server-Antwort: CONTENT oder NOSUCHFILE
Eingabe Client: anderer Inhalt
```

Im Hintergrund passiert nicht anderes, als dass der Dateiinhalt mit dem neuen Inhalt überschrieben wird.

2.8 Datei löschen

Um eine Datei zu löschen kann der Befehl "DELETE“ verwendet werden. Damit können nur einzelne Dateien gelöscht werden.

```
Befehl: DELETE Dateiname
Beispiel:
Eingabe Client: DELETE testfile
Server-Antwort: DELETED oder NOSUCHFILE
```

3 Planung / Vorbereitung

In diesem Kapitel wird der geplante Lösungsweg beschrieben. Da erst im Verlauf der Arbeit der entsprechende Stoff in der Vorlesung „Systemprogrammierung“ behandelt wurde, entstand der Lösungsweg nicht am Anfang, sondern fortlaufend. Grundlegend wurden 3 Varianten geprüft.

3.1 Variante 1: Shared Memory pro Datei und Double Linked List für Kontrollstruktur

Als Kontrollstruktur ist eine Double Linked List vorgesehen. Jede Datei hat einen Eintrag und hat einen Nachfolger und einen Vorgänger, welche als Pointer im eigenen Listenelement gespeichert sind. So ist es möglich, bei der Grösse der Struktur sehr flexibel zu sein, es müsste keine Limite (ausser Grösse der Pointer) beachtet werden. Allerdings ist einiger Zusatzaufwand nötig, um diese Liste zu pflegen. Der Inhalt der Datei selber liegt in einem Shared-Memory-Bereich, dessen ID ebenfalls im Listenelement gespeichert wird. Wenn ein Prozess den Inhalt einer Datei braucht, so kann er das Shared-Memory-Segment anbinden. Das Locking geschieht mit Semaphoren, allerdings würde der Locking-Zustand ebenfalls im Listenelement gespeichert und bei Bedarf einem Semaphor zugewiesen.

3.2 Variante 2: Shared Memory pro Datei und Array als Kontrollstruktur

Wie bei der Variante 1 soll der Inhalt der Datei in einem Shared-Memory-Segment pro Datei liegen. Die Informationen (Länge, Dateiname, Shared-Memory-ID) werden in einem Struct-Array abgelegt. Sobald Informationen gesucht werden müssen, muss halt jeweils durch das Array iteriert werden. Die maximale Anzahl Dateien ist so bei der Initialisierung des Array vorgegeben und kann nur mühsam geändert werden. Falls ein neues Array in einem grösseren Shared-Memory-Bereich erstellt würde, müsste an alle Child-Prozesse der neue Shared-Memory-Bereich mitgeteilt und beim Kopieren die ganzen Dateiinformationen gelockt werden. Das Locking wird über ein Set von Semaphoren erledigt, welches zu Beginn die Anzahl Dateien + 1 Semaphoren enthält. Der Überschüssige Semaphor wird für ein Locking des Arrays verwendet und ist nur vorsorglich erstellt, falls es später einen Anwendungszweck dafür gibt. Die Nummerierung im Array und im Semaphor-Set deckt sich, das heisst Datei 1 auf array[0] hat im Semaphor-Set den Semaphor mit der Nummer 0.

3.3 Variante 3: Nur eine Struktur

Bei der dritten Variante wird alles in einem Array gespeichert, welches in einem Shared-Memory-Bereich liegt. Sowohl die Informationen zur Datei als auch die Inhalte. Der Nachteil ist, dass für jede Fix gleich viel Speicher reserviert wird, und dadurch nicht optimal ist hinsichtlich der Speichereffizienz (gespeicherte Daten vs. Benötigter Speicherplatz). Das Locking wird wiederum über Semaphore erledigt, wie bei der Variante 2.

3.4 Variantenentscheid

Als Neuling auf dem Gebiet fiel es mir nicht leicht, mich für eine Variante zu entscheiden. Aus diesem Grund wurden zu den verschiedenen Themen diverse kleine Test gemacht, um den Aufwand einigermassen abschätzen zu können. Die Variante 1 wurde als zu aufwendig verworfen und die Variante 2 als beste Variante mit tragbarem Aufwand eingschätzt.

3.5 Thread oder Fork

Am Anfang war aus der Vorlesung nur die Methode mit parallelen Prozessen bekannt, also wurde hier schon mal auf fork() gesetzt. Dazu beigetragen hat sicher auch, dass fork() und Shared Memory vom Dozenten empfohlen wurden. Spezielle Gründe dafür gibt es nicht, allerdings hat sich gezeigt, dass das Memory-Handling mit pthreads wohl einfacher ausgefallen wäre.

3.6 Kontrollstruktur

Als Kontrollstruktur wird ein Array verwendet, welches bei Programmstart initialisiert wird. Informationen zu jeder abgelegten Datei werden in der Kontrollstruktur abgelegt, dies umfasst mindestens Dateiname, Länge der Datei und Speicherort.

3.7 Speichern der Datei

Der Inhalt der Datei selber soll in einem Shared-Memory-Segment legen. Für jede Datei wird ein solches erstellt verwaltet. Dies soll dem Programm ermöglichen, nur so viel Speicherplatz zu verwenden wie die Dateien auch benötigen. Ebenfalls ist so gewährleistet, dass nicht mit einer fixen Dateigrösse pro Datei gearbeitet werden muss. Jede Datei liegt in einem Speicherbereich, der genau so gross ist wie der Inhalt der Datei.

3.8 Locking-Mechanismus

Um die entsprechenden Dateien bei gleichzeitigen Zugriffen sperren zu können, wird mit Semaphoren gearbeitet. Initial soll ein Set von Semaphoren erstellt werden, welches pro

Datei einen Semaphor enthält und einen zusätzlichen für die Kontrollstruktur. Die Kontrollstruktur muss ebenfalls gesperrt werden können, ansonsten kann der Fall auftreten dass zwei Prozesse gleichzeitig Informationen schreiben wollen. So ist für jede Datei ein Semaphor in diesem Set enthalten und kann auf Dateiebene gesperrt werden.

3.9 Kommunikation

Die Netzwerkkommunikation soll über TCP/IP-Sockets geschehen. Unix Domain Sockets sind nicht bekannt und waren auch nicht zur Genüge in einer Vorlesung behandelt, so fiel diese Variante schon mal ganz sicher weg.

4 Probleme / Wege / Lösungen

In diesem Kapitel werden einige Einzelheiten der Arbeit genauer beleuchtet sowie Probleme diskutiert. Gestartet wurde mit der Absicht, die Variante zwei aus dem Kapitel 3 umzusetzen, was sich aber als problematisch heraus stellte. Die abgelieferte Lösung entspricht nicht dieser Variante, sondern wurde schlussendlich in die Variante 3 abgeändert.

4.1 Shared Memory pro Datei

Die Idee, für jede Datei einen Shared Memory-Bereich zu verwenden, war eigentlich die Nächstliegendste. Bei den ersten Tests in einem Testprogramm hat eigentlich alles gut geklappt: Shared Memory-Bereiche konnten erstellt und an verschiedene Adressbereiche von Prozessen gemappt werden. Als dann aber die Implementation im eigentlichen Server-Programm anstand tauchten sehr seltsame Phänomene auf. Nachdem es anfänglich geklappt hatte, funktionierte es wie von Zauberhand nicht mehr, auch Neustarts und eine neue Kompilation funktionierten nicht. Die errno-Nachricht war "No space left on device"was aber nicht stimmen konnte, da mit anderen Programmen und Prozessen problemlos Shared Memory-Bereiche erstellt werden konnten.
Nachdem die Implementation eigentlich nur noch an diesem Punkt scheiterte, habe ich mich dazu entschieden alles auf Variante 3 umzuprogrammieren. Die Lösung auf dem ursprünglich angedachten Weg schien zwar nah, aber nach 15 Stunden Fehlersuche ohne Resultat und einer näher Rückenden Abgabefrist war diese Änderung unabdingbar, auch wenn sie nochmal einen erheblichen Aufwand bedeutet hat.

4.2 Semaphore - zu viele

Es hat sich gezeigt, dass das vorhandene Wissen nicht ganz mit der Realität übereingestimmt hat. Ich hatte einen grossen AHA-Effekt, als ich beabsichtigt habe für jedes File einen Semaphor zu erstellen. Das ging eigentlich auch gut, nur habe ich statt eine bestimmte Anzahl Semaphoren eine bestimmte Anzahl Semaphoren-Sets erstellt. Der Ausschnitt aus der MAN-Page von semget() hilft weiter:

```
The semget() system call returns the System V semaphore set
identifier associated with the argument key.  A new set of nsems
semaphores is created if key has the value IPC_PRIVATE or if no
existing semaphore set is associated with key and IPC_CREAT is
specified in semflg.
```

Nachdem ich das dann Begriffen habe, sind dann auch die Fehlermeldungen verschwunden, wonach die maximale Anzahl zulässiger Semaphore Arrays erreicht sei. Auf dem verwendeten System (Ubuntu 14.04 LTS) sind die Standardwerte wie folgt gesetzt:

```
------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767
```

Das Handling der Semaphore geschah dann mit semop(). Im Code sieht das wie folgt aus, wobei i der Index für die derzeit verwendete Datei ist, sowohl im Array wo die Dateien organisiert sind als auch im Semaphor-Set.

```
//lock
sem_buf_temp = sem_buf_lock;
sem_buf_temp.sem_num = i;
retcode = semop(sem_fh_id, &sem_buf_temp,1);

//unlock
sem_buf_temp = sem_buf_unlock;
sem_buf_temp.sem_num = i;
retcode = semop(sem_fh_id, &sem_buf_temp,1);
```

5 Fazit

Im Laufe dieser Arbeit habe ich sehr vieles gelernt. Obwohl ich auch zugestehen muss, dass ich den Zeitrahmen für eine Seminararbeit wohl massiv übertroffen habe. Es war ein interessanter Ausblick in eine "Welt" die ich zwar tagtäglich einsetze, aber kaum je hinter die Kulisse sehe. Die Vorstellung, wie aufwändig und Komplex es ist Systemnahe Software zu schreiben, geschweige denn ein Teil eines Betriebssystems, habe ich erst jetzt. Wenn ich schon nur daran denke, was nun ein Dateisystem alles abdecken muss in den Grundfunktionen, und dann kommen noch Performance-Anforderungen, Verschlüsselung, Deduplikation, Fehlertoleranz etc. dazu!
Aufgrund der fehlenden Erfahrung habe ich wohl viele Anfängerfehler gemacht und es sind bestimmt noch viele Unschönheiten und Fehler im Code versteckt. Was ich ändern würde ist, dass ich die Teilprobleme noch mehr unterteile und eines nach dem anderen sauber teste und in den Hauptcode implementiere. Um Überhaupt einen Einblick zu erhalten hatte ich damit begonnen, alle möglichen Funktionen noch einmal zu testen und dann mehr oder weniger geordnet in den Programmcode zu übernehmen. Das hat es bei der Fehlersuche nicht immer einfach gemacht. Ebenfalls habe ich Mühe damit bekundet, Fehler richtig zu debuggen. In diesem Bereich war ich oft ratlos und habe mit unzähligen printf() von Variabeln versucht den Fehler zu umzingeln. Ich bin mir sicher, dass sich in diesem Bereich noch sehr viel lernen lässt.
Schade fand ich, dass das notwendige Wissen erst parallel zur Seminararbeit im Unterricht bei Herrn Brodowsky aufgebaut wurde. Für mich als Neuling war der Unterricht sehr viel Wert, vielleicht gibt es ja die Möglichkeit den Unterricht von Herrn Brodowsky in ersten Teil des Semesters zu verdichten und dafür das Seminar erst auf den zweiten Teil zu lancieren? Mir hätte es wohl geholfen.