

# Concurrent Fileserver Dokumentation

Klassifizierung  
Datum  
Autor(en)  
Version  
Status

Allgemein zugängliche Informationen  
03. Juni 2014  
Milutinovic Milorad  
1.0  
Freigegeben

# Inhaltsverzeichnis

<b>1.</b>	<b>Einleitung .....</b>	<b>3</b>
1.1	Ausgangslage .....	3
1.2	Ziel der Arbeit .....	3
1.3	Aufgabenstellung .....	3
1.4	Erwartete Resultate .....	3
1.5	Zeitplan .....	4
1.6	Seminarrelevante Daten .....	4
<b>2.</b>	<b>Projekt .....</b>	<b>5</b>
2.1	Konzept .....	5
2.2	Protokoll .....	5
2.3	Server .....	6
2.3.1	Funktion List .....	7
2.3.2	Funktion Create .....	7
2.3.3	Funktion Read .....	7
2.3.4	Funktion Update .....	7
2.3.5	Funktion Delete .....	8
2.4	Client .....	8
2.5	Probleme und Lösungen .....	8
2.5.1	Implementation des Protokolls .....	8
2.5.2	Verwaltung von den Files .....	8
2.6	Testszenarien .....	8
2.7	Software download .....	9
2.8	Software kompilieren .....	9
2.9	Benutzeranleitung .....	10
<b>3.</b>	<b>Schlusswort .....</b>	<b>11</b>
3.1	Weiterentwicklungsmöglichkeiten .....	11
3.2	Fazit .....	11
<b>4.</b>	<b>Anhang .....</b>	<b>12</b>
4.1	Quellen .....	12
4.2	Tabellenverzeichnis .....	12

# 1. Einleitung

## 1.1 Ausgangslage

Im Rahmen des Seminars «Concurrent C» konnte ich zwischen zwei Themen wählen: «Fileserver» oder einem «Multi-User-Editor». Ich habe mich für den Fileserver entschieden.

## 1.2 Ziel der Arbeit

Das Ziel ist «Concurrent programming» zu verstehen und einen auf Linux lauffähigen Fileserver in C zu erstellen.

## 1.3 Aufgabenstellung

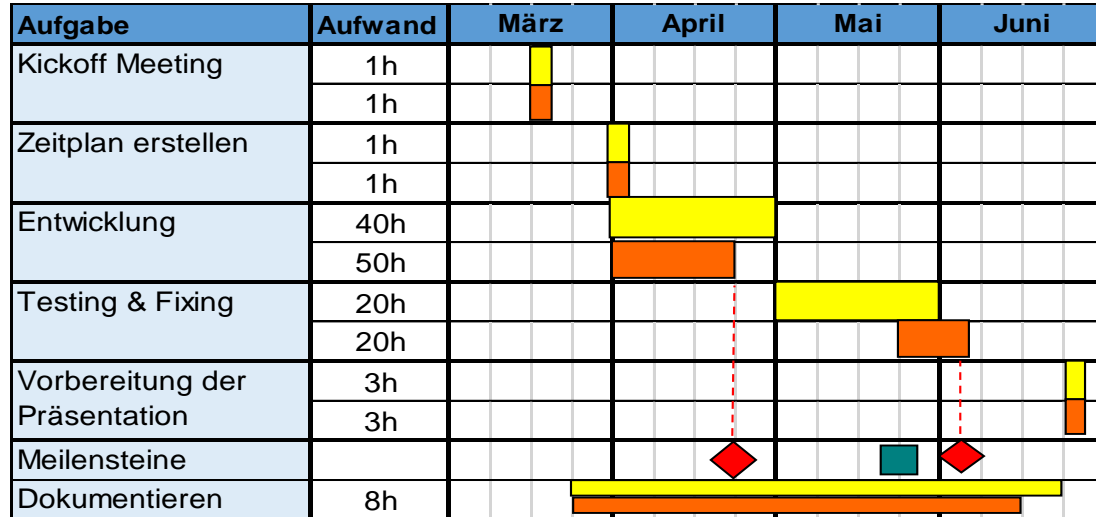
- Dateien sind nur im Speicher vorhanden
- Das echte Dateisystem darf NICHT benutzt werden
- Mehrere gleichzeitige Clients möglich
- Lock auf Dateiebene (kein globaler Lock !)
- Kommunikation via TCP/IP
- Einsatz von shm und fork
- Vordefiniertes Protokoll implementieren
- Alle Indeces beginnen bei 0
- Debug-Ausgaben von Client/Server auf stderr
- Wenn die Eingabe valid ist, bekommt der Client ein OK
- Locking, gleichzeitiger Zugriff im Server lösen
- Client muss \*nie\* retry machen

## 1.4 Erwartete Resultate





- Lauffähiger Fileserver unter Linux
- Dokumentation des Seminars
- Präsentation

## 1.5 Zeitplan

Effektiver Aufwand: ~85 Stunden



Legende:

	Soll Zeit
	Ist Zeit
	Meilenstein
	Code Review

## 1.6 Seminarrelevante Daten

12.03.2014	Kickoff
16.03.2014	Angabe git repository
12.04.2014	Zwischenstand & EBS-Eintrag
18.04.2014	EBS Eintrag abgeschlossen
22.06.2014	Abgabe Arbeit

**Tab. 1:** Seminarrelevante Daten

## 2. Projekt

### 2.1 Konzept

Der Fileserver wird mit der Programmiersprache C implementiert. Das Projekt hat folgende Verzeichnis Struktur, welche für die Abstraktion des Codes genutzt wird:

include/	Definitionen von Funktionen und Variablen.
lib	Hier werden .c Files abgelegt, welche von run.c oder test.c gebraucht werden.
run.c	Das File run.c implementiert den Fileserver. Hier wird werden alle Clientanfragen interpretiert und entsprechend bearbeitet.
test.c	Das File test.c implementiert den Fileclient. Der Client sendet eine Anfrage an den Server, wartet auf die Antwort des Servers und gibt diese schliesslich aus.
Makefile	Das Makefile gibt an was kompiliert werden soll. Es dient als Datenbasis für das Unix-Programm namens «make».
README.md	Beinhaltet die Informationen wie das Projekt kompiliert und genutzt werden kann.

**Tab. 2:** Konzept

### 2.2 Protokoll

Das Protokoll ist vorgegeben und definiert 5 verschiedene Funktionen, welche zu implementieren sind. Folgende Tabelle beschreibt diese 5 Funktionen näher.

<u>List:</u> Client sendet: LIST\n Server antwortet: ACK NUM_FILES\n FILENAME\n FILENAME\n FILENAME\n ... Server Beispiel: ACK 3 abc def aei	<u>Create:</u> Client sendet: CREATE FILENAME LENGTH\n CONTENT Client Beispiel: CREATE abc 6\n Hello\n Server antwortet: FILEEXISTS\n oder FILECREATED\n	<u>Read:</u> 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
<u>Update:</u> Client sendet: UPDATE FILENAME LENGTH\n CONTENT Client Beispiel: UPDATE abc 3\n 12\n Server antwortet: NOSUCHFILE\n oder UPDATED\n	<u>Delete:</u> Client sendet: DELETE FILENAME\n Client Beispiel: DELETE abc\n Server antwortet:  NOSUCHFILE\n oder DELETED\n	

**Tab. 3:** Das Vordefinierte Protokoll

## 2.3 Server

Die Aufgabe des Servers ist die Anfragen des Clients zu bearbeiten und die Dateien «Shared Memory» zu verwalten. Damit der Server im Hintergrund läuft, wird zuerst ein Daemon Prozess erzeugt. Nach dem Serverstart erhält der Benutzer wieder den Zugriff auf seine Shell und kann weitere Aufgaben erledigen. Die Prozess ID des Daemon Prozesses wird in der Datei «run.pid» festgehalten. Diese wird für das Stoppen des Servers benötigt. Nun wird ein Bereich im «Shared Memory» initialisiert, der Platz für 100 Dateien bietet. Eine Datei entspricht folgender Struktur:

1	struct storedef {
2	sem_t sem; //Semaphor für das File
3	int semid; //Semaphor ID
4	int shmid; //Shared Memory ID
5	unsigned long long filesize; //Dateigrösse
6	char filename[64]; //Dateiname
7	char *content; //Dateinhalt
8	int shmidcontent; //Shared Memory ID für die Datei
9	int state; //Zeigt an, ob die Struktur durch eine Datei belegt ist
10	// 0 = nicht belegt, 1 belegt
11	};

**Tab. 4:** Definition einer Struktur

Während der Initialisierung des Speichers für die Dateien wird für jede Struktur ein Semaphor angelegt und der Status auf 0 gesetzt. Jede Struktur erhält auch einen eigenen «Shared Memory» Segment, welcher von den Kind Prozessen genutzt wird. Diese 100 Strukturen werden in einem Array aufbewahrt. Im weiteren Verlauf der Dokumentation nenne ich dieses Array Storage. Falls 80% vom Storage besetzt sind, wird er vom nächsten Kind Prozess vergrössert. Leider bietet «Shared Memory» nicht die Möglichkeit ein bestehendes «Shared Memory» Segment zu vergrössert. Deshalb wird ein neuer «Shared Memory» Segment angelegt, welcher den Platz des vorherigen Segments verdoppelt. Das heisst, dass nach der ersten Vergrösserung Storage den Platz für 200 Strukturen bzw. Dateien bietet. Alle Informationen werden dann vom alten in den neuen Storage kopiert. Damit sich hier die Kinder Prozesse nicht in die Quere kommen und kein Datenverlust entsteht, wird ein Semaphor eingesetzt, um den kritischen Bereich zu schützen. Nun werden die alten «Shared Memory» Segmente gelöscht und die Kinder Prozesse können mit dem neuen Storage arbeiten.

Nach der Initialisierung des Speichers für die Dateien, kann der Server die Client-Anfragen akzeptieren. Bei einer Clientanfrage generiert der Server einen neuen Kind Prozess, der sich genau um diese eine Anfrage kümmert während der Eltern Prozess wieder in den Zustand «Listening» geht. Der erstellte Kind Prozess bearbeitet dann effektiv die Anfrage des Clients. Der neue Kind Prozess tritt in eine Endlosschleife ein bis eine Abbruchbedingung erfüllt ist. Folgende Tabelle zeigt die Abbruchbedingungen der jeweiligen Funktionen an:

Funktion	Abbruchbedingung
List,read,update,delete	Nach dem ersten «\n» - Zeichen wird die Endlosschleife verlassen
create,update	Der Server erhält die Grösse der zu erstellenden oder der zu aktualisierenden Datei. Nach dem ersten Zeichen «\n» zählt der Server die Anzahl der erhaltenen Bytes. Sobald diese Zählung der mitgelieferten Dateigrösse entspricht, wird die Endlosschleife verlassen.

**Tab. 5:** Abbruchbedingungen

### 2.3.1 Funktion List

Der neu erstellte Kind Prozess erhält den String «list\n». Die Methode «listFiles(int clientSocket)» iteriert durch den Storage und sendet die Anzahl der verfügbaren Dateien sowie die Dateinamen zurück an den Client.

### 2.3.2 Funktion Create

Nach dem Erhalt der Daten speichert der Server die neue Datei temporär in den dynamischen Speicher, auch Heap genannt. Nun wird durch den Storage iteriert und die erste freie Struktur im Storage wird genutzt, um die Datei abzulegen. Um den genauen Ablauf zu erklären benutze ich den folgenden Code-Ausschnitt:

1	for (i=0; i < getStorageSize(); i++){
2	address = storage[i];
3	addr = (struct storagedef *) shmat(address.shmid, NULL, 0);
4	sem_wait(&addr->sem);
5	if (addr->state == 0){
6	fexist = FileExists(filename);
7	if (fexist == 1){
8	sem_post(&addr->sem);
9	rc = shmdt(addr);
10	rc_check(rc, "28-shmdt() failed!");

**Tab. 6:** Code Ausschnitt – Kritischer Bereich

Nach dem der Kind Prozess das «Shared Memory» Segment einer Struktur in Anspruch genommen hat, nimmt er den Semaphor der Struktur. Falls kein anderer Prozess den gleichen Semaphor beansprucht, wird Zeile 5 ausgeführt. Ansonsten wartet der Prozess auf der Zeile 4 solange bis der andere Prozess den Semaphor freigegeben hat. Somit wird mit der Zeile 5 sichergestellt, dass die aktuelle Struktur wirklich frei ist. In diesem Fall wird für den Dateinhalt ein neuer «Shared Memory» Segment angelegt, sodass die Daten aus dem Heap in das «Shared Memory» kopiert werden können. Schliesslich wird das Attribut «state» der Struktur auf 1 gesetzt und der Client über die Erstellung der Datei informiert.

### 2.3.3 Funktion Read

Der Storage wird nach dem verlangten Dateinamen untersucht. Falls eine Struktur mit dem angegebenen Dateinamen gefunden wurde, werden dessen Inhalt sowie der Name zusammen mit der Dateigrösse an den Client gesendet.

### 2.3.4 Funktion Update

Update funktioniert wie «create». Der Unterschied ist, dass das bestehende «Shared Memory» Segment des Dateiinhaltes zuerst gelöscht und mit der neuen Grösse neu angelegt wird. Anschliessend wird der Client über die Aktion informiert.

### 2.3.5 Funktion Delete

Die Funktion Delete löscht das «Shared Memory» Segment und setzt das State-Attribut der Struktur auf 0. Sollte die zu löschende Datei gar nicht existieren, so wird der Client auch darüber informiert.

## 2.4 Client

Der Client baut zuerst eine Verbindung zum Server Socket auf. Abhängig von den Argumenten sendet er eine Nachricht an den Server und wartet auf dessen Antwort. Die Antwort wird danach ausgegeben.

## 2.5 Probleme und Lösungen

### 2.5.1 Implementation des Protokolls

Ich habe die Implementation des Protokolls falsch verstanden. Nach einem Review von Herr Schottelius wurde ich darauf aufmerksam gemacht, sodass ich die Implementation des Protokolls überarbeiten konnte.

### 2.5.2 Verwaltung von den Files

Den Storage habe ich zuerst im Heap abgelegt, weil die Vergrößerung des Speichers im Heap einfach mit der Funktion realloc() gemacht werden kann. Jedes Mal als ein neuer Prozess erstellt wurde, hat der Eltern Prozess geschaut, ob der Storage über 80% besetzt war und hat dann eine Vergrößerung vorgenommen. Das Problem war dann, dass Clients während dieser Vergrößerung keine Verbindung mit dem Server aufbauen konnten. Um dies zu verhindern, habe ich den Storage in das Shared Memory verlagert, sodass Kind Prozesse die Vergrößerung vornehmen können.

## 2.6 Testszenarien

Datei erstellen	✓
Datei löschen	✓
Datei aktualisieren	✓
Datei lesen	✓
Dateien anzeigen lassen	✓
100 Dateien erstellen	✓
200 Dateien erstellen. 100 Dateien im ersten und 100 im zweiten Fenster.	✓
Nach dem Stoppen des Servers sind alle «Shared Memory» Segmente gelöscht	✓

**Tab. 7:** Testszenarien



## 2.7 Software download

Folgende Tabelle zeigt an wie die Software von github heruntergeladen werden kann.

```
loki@svloki:/var/tmp$ mkdir blub
loki@svloki:/var/tmp$ cd blub
loki@svloki:/var/tmp/blub$ git clone https://github.com/Milorad/concurrent_filetransfer.git
Cloning into 'concurrent_filetransfer'...
remote: Reusing existing pack: 125, done.
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 141 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (141/141), 37.57 KiB, done.
Resolving deltas: 100% (62/62), done.
loki@svloki:/var/tmp/blub$ pwd
/var/tmp/blub
loki@svloki:/var/tmp/blub$ ls -ltrh
total 4.0K
drwxr-xr-x 5 loki school 4.0K May 31 19:34 concurrent_filetransfer
loki@svloki:/var/tmp/blub$ cd concurrent_filetransfer/
loki@svloki:/var/tmp/blub/concurrent_filetransfer$ ls
include lib Makefile README.md run.c test.c
loki@svloki:/var/tmp/blub/concurrent_filetransfer$
```

## 2.8 Software kompilieren

Für das Kompilieren werden die Programme make und ar benötigt. Zuerst werden die Files im Verzeichnis lib kompiliert und danach werden die Object Dateien zu einem Archiv namens «lib/genlib.a» zusammengefasst. Dieses Archiv, auch statische Library genannt, wird für die Kompilierung des Clients und des Servers benötigt. Im Gegensatz zu den «shared Libraries» werden die «static Libraries» ein Teil von den ausführbaren Dateien. Wenn «shared Libraries» eingesetzt werden, so werden Sie nicht während der Kompilierung in die ausführbaren Dateien eingebunden sondern erst zur Laufzeit.

```
loki@svloki:/var/tmp/blub/concurrent_filetransfer$ make
gcc -c -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib lib/tcp-client.c -o lib/tcp-client.o -lpthread -lrt -lm
gcc -c -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib lib/tcp-server.c -o lib/tcp-server.o -lpthread -lrt -lm
gcc -c -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib lib/clientlib.c -o lib/clientlib.o -lpthread -lrt -lm
gcc -c -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib lib/genlib.c -o lib/genlib.o -lpthread -lrt -lm
ar crs lib/genlib.a lib/tcp-client.o lib/tcp-server.o lib/clientlib.o lib/genlib.o
gcc -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib run.c lib/genlib.a -o run -lpthread -lrt -lm
gcc -Werror -Wall -g -O2 -std=gnu99 -I./include -L./lib test.c lib/genlib.a -o test -lpthread -lrt -lm
loki@svloki:/var/tmp/blub/concurrent_filetransfer$
```

## 2.9 Benutzeranleitung

Nach der Kompilierung muss zuerst der Server gestartet werden.

```
# ./run start
```

Danach kann der Client wie folgt eingesetzt werden:

Beispiel	Beschreibung
# ./test create /etc/passwd file created #	Eine Datei in das «Shared Memory» ablegen.
# ./test list ACK 1 /etc/passwd #	Alle verfügbaren Dateien auflisten.
# ./test read /etc/passwd FILECONTENT /etc/passwd 1587 root:x:0:0:root:/root:/bin/bash ... #	Den Inhalt einer Datei anzeigen.
# ./test update /etc/passwd /etc/group updated #	Den Inhalt der Datei /etc/passwd mit dem Inhalt der Datei /etc/group ersetzen.
# ./test delete /etc/passwd deleted #	Die Datei löschen.

**Tab. 8:** Benutzungsbeispiele

Der Server kann mit dem Kommando «./run stop» gestoppt werden.

### 3. Schlusswort

#### 3.1 Weiterentwicklungsmöglichkeiten

- Unterstützung von binären Dateien
- Datei umbenennen
- Login Funktion und Ablage der Dateien auf das Filesystem

#### 3.2 Fazit

Dies war mein erstes C-Programm. Deshalb war zu erwarten, dass ich mehr Zeit als vorgesehen aufwenden musste, um eine funktionierende Applikation zu erstellen. Während der Durchführung des Seminars konnte ich endlich einige Sachen wirklich verstehen. Leider konnte ich auch feststellen, dass die Programmierung mit C sehr mühsam sein kann. Vor allem die Untersuchung von so genannten «segmentation faults». Eventuell könnte ein grösseres Projekt realisiert werden indem die Arbeit in verschiedene Gruppen unterteilt wird.

#### 3.3 Überprüfung der Aufgabenstellung und der Resultate

Alle erwähnten Punkte in der Aufgabenstellung wurden erfüllt. Die erwarteten Resultate sind auch vorhanden.

## 4. Anhang

### 4.1 Quellen

<http://beej.us/> c Beispiele

<https://github.com/bk1> Beispiele aus dem Unterricht

### 4.2 Tabellenverzeichnis

Tab. 1	Seminarrelevante Daten	Seite 4
Tab. 2	Konzept	Seite 5
Tab. 3	Das Vordefinierte Protokoll	Seite 5
Tab. 4	Definition einer Struktur	Seite 6
Tab. 5	Abbruchbedingungen	Seite 6
Tab. 6	Code Ausschnitt – Kritischer Bereich	Seite 7
Tab. 7	Testszenarien	Seite 9
Tab. 8	Benutzungsbeispiele	Seite 11