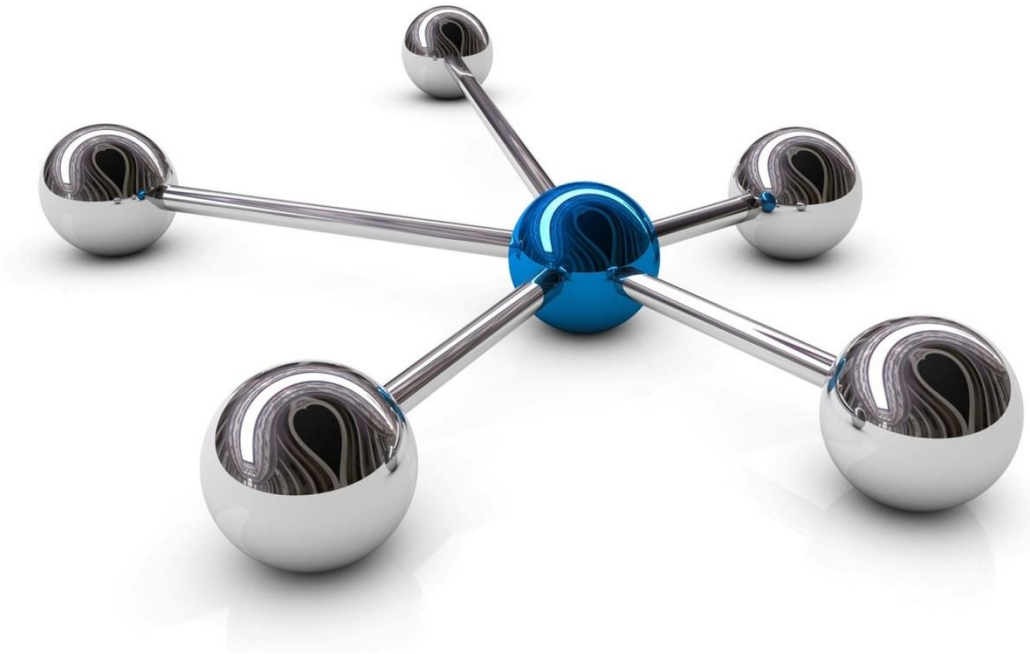
Seminararbeit 2014

Concurrent C Programming
Multi-User-Editor



Verfasser	Steve Heller
Betreuer	Nico Schottelius
Schulort	ZHAW
Erscheinungsjahr	2014

Inhaltsverzeichnis

1 Themenwahl

1.1 Eigenmotivation

Nachdem uns von Herrn Nico Schottelius am Seminar Kick-Off der grobe Themenbereich *Concurrent C* vorgestellt wurde, hatten wir die Auswahl zwischen zwei Projekten:

- Fileserver
- Multi-User-Editor

Mich hat das zweite Projekt – also der *Multi-User-Editor* – auf anhieb angesprochen und ein erstes Grobkonzept begann sich vor meinen Augen aufzubauen.

Das Projekt des *Multi-User-Editor* war besser auf meine bereits erlernten Fähigkeiten in der Programmiersprache C zugeschnitten. Als Fan von klarem und modularem Code wollte ich diesen Editor so einfach aber vielseitig wie möglich halten.

1.2 Abgrenzung der Umsetzung

Im Rahmen der Umsetzung musste ich leider schnell feststellen, dass die Lösung doch nicht so trivial ist, wie ursprünglich angenommen. Die Umsetzung benötigte viel mehr Aufwand und die investierte Zeit überstieg das vorgegebene Budget rasch.

Aus diesem Grund wurde ein Teil der Funktionalität weggelassen. Der *Multi-User-Editor* erfüllt zwar alle aus der Aufgabenstellung hervorgehenden Anforderungen – aber er führt keine Datei-Modifikationen aus. Dies bedeutet, immer dort, wo der Server eine Zeile in einer Datei bearbeiten soll, wird stattdessen nichts ausgeführt.

Zur einfachen Handhabung und späteren Erweiterung des Programms sind diese Zeilen im Code mit folgendem Kommentar gekennzeichnet:

```
// here comes the file modification
```

1.3 Zielgruppe

Dieses Dokument richtet sich hauptsächlich an Interessenten meiner Seminararbeit – genauer Dozenten und Kommilitonen. In diesem Sinne wird ein Grundverständnis der Programmierung sowie der Sprache C vorausgesetzt.

2 Aufgabenstellung

Das Ziel dieses Seminars sollte es sein, die Problematik der Concurrency in der Informatik zu analysieren und mögliche Lösungen am Beispiel eines Programmes umzusetzen.

Concurrency in der Informatik bedeutet, dass ein System oder ein Programm mehrere Befehle gleichzeitig ausführen kann. Dabei können die einzelnen Befehle miteinander kommunizieren – beispielsweise Zwischenergebnisse austauschen.

2.1 Multi-User-Editor

Bei dieser Veranschaulichung von Concurrency-Lösungswegen handelt es sich um eine Server-Client Applikation auf TCP/IP Basis. Der Server erwartet dabei eine Menge von Befehlen und reagiert entsprechend darauf.

Wichtig ist eine gleichzeitige Behandlung von mehreren Anfragen. Dies bedeutet, es müssen sich mehrere Benutzer gleichzeitig mit dem Serververbinden und Anfragen ausführen können.

Im Rahmen des *Multi-User-Editors* bedeutet dies, dass ein entsprechendes Locking zum Einsatz kommen muss. Vorgabe hierbei war, dass das Locking nur auf Zeilenebene erlaubt ist und nicht das ganze Dokument gesperrt werden darf.

Die genaue Aufgabenstellung befindet sich im GitHub Repository[1] von Herrn Nico Schottelius

3 Grobkonzept

In einem ersten Grobkonzept wurden die benötigten Elemente sowie mögliche Ansatzwege eruiert. Die dabei getroffenen Entscheidungen sollten in der späteren Umsetzung einfliessen.

3.1 Verbindungsaufbau und Prozesshandling

Damit eine Client-Server Verbindung via TCP/IP aufgebaut werden kann, muss Serverseitig auf eine entsprechende Verbindungsanfrage seitens Client gewartet werden. Dies soll im Hauptprozess des Servers geschehen.

Bei jeder eingehenden Verbindung soll der Server dann einen neuen Child-Prozess starten und die weitere Kommunikation mit dem Client über diesen Prozess abwickeln. Dies bedeutet, pro Client-Server Verbindung ist ein eigener Prozess vorhanden. Darüber hinaus wird im Hauptprozess jeweils auf eine neue Verbindungsanfrage gewartet.

Beim Disconnect eines Clients soll der Prozess jeweils sauber beendet werden.

3.2 Lockingmechanismen im Shared Memory

Da jeder Prozess seinen eigenen Speicherbereich im Memory zugewiesen bekommt, muss ein Locking übergeordnet passieren. Die beste Wahl dafür ist eine Speicherung im Shared Memory.

Das Shared Memory ist dabei ein gemeinsam genutzter Speicherbereich im Memory und kann von allen Prozessen gelesen und beschrieben werden.

Wichtig beim Shared Memory ist eine entsprechende Absicherung via Semaphore, damit keine Race Condition auftritt.

3.3 Das Locking im Detail

Die ursprüngliche Idee war, im Shared Memory ein Array anzulegen, welches jeweils die gesperrten Zeilen enthält. Dabei wird bei der Sperrung an das Ende des Arrays die Zeilennummer angehängt. Wird eine Zeile benötigt, wird das ganze Array durchlaufen und verglichen, ob die entsprechende Zeilennummer im Array vorhanden ist – in diesem Fall wäre die Zeile gesperrt und ein Zugriff nicht möglich.

Nach einiger Recherchen kam mir jedoch eine bessere Idee:
Das Array kann direkt auf die Zeilen umgemünzt werden können. Dies bedeutet, die erste Position im Array entspricht der ersten Zeile im File, die zweite Position im Array entspricht der zweiten Zeile im File, und so weiter. Im Array steht dann an der entsprechenden Position eine Null oder eine Eins – wobei die Null einer Freigabe und die Eins einer Sperrung entspricht.

Soll nun beispielsweise auf die Zeile 21 zugegriffen werden, muss nicht das ganze Array durchlaufen und nach dem Integer 21 gesucht werden, sondern es kann direkt der Wert an Position 20[1] ausgelesen werden – steht dort eine Null ist die Zeile zugreifbar, steht eine Eins muss der Client warten.

4 Umsetzung im Detail

Nachdem das Grobkonzept stand, ging es an die Umsetzung. Im folgenden Kapitel sind die wichtigsten Milestones chronologisch nach ihrer Implementation beschrieben.

4.1 Grundlegende Serverfunktionen

Als erster Schritt war eine einfache Client-Server Applikation nötig. Diese beinhaltete keine grosse Funktionalität, sondern sollte lediglich als Projektstart dienen. Die zukünftigen Produktionsschritte sollten darauf basieren und damit getestet werden können.

Der Server erwartet nach dem Start einen Verbindungsaufbau seitens Client:

```
listen(sock, 5);
int clientSock = accept(sock, (struct sockaddr *) & clientAddress,
&clientAddressLen);
```

[1] Indizes beginnen bei 0.

Sobald der Client einen Verbindungsaufbau initiiert hat, gerät der Server in einen endlosen Loop und wartet auf Texteingaben seitens Client:

```
while (breakUp == 0) {
  int clientSock = accept(sock, (struct sockaddr *) & clientAddress,
&clientAddressLen);
  bzero(buffer,256);
  read(clientSock,buffer,255);
  if (strcmp(buffer,"EXIT") == 0) {
    printf("Client disconnected\n");
    breakUp = 1;
  } else {
    printf("Client wrote: %s\n", buffer);
  }
  close(clientSock);
}
```

Die Texteingaben, welche der Server erhält, werden jeweils auf der Kommandozeile ausgegeben. Erhält der Server die Zeichenkette *EXIT* so bricht er den Loop ab und beendet sich.

Der Client funktioniert analog dazu, jedoch sendet dieser natürlich eine Verbindungsanfrage, statt darauf zuwarten.

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock, (struct sockaddr *) &serverAddress, sizeof(serverAddress));
```

Ausserdem wird im Loop auf eine Texteingabe seitens Benutzer gewartet, welche dann an den Server übertragen wird:

```
while (breakUp == 0) {
  char input_string[256];
  char *p;
  unsigned int inputStringLen;
  fgets(input_string, sizeof(input_string), stdin);
  if ((p = strchr(input_string, '\n')) != NULL) {
    *p = '\0';
  }
  if (strcmp(input_string,"EXIT") == 0) {
    printf("You disconnected!\n");
    breakUp = 1;
  }
  inputStringLen = strlen(input_string);
  int count = send(sock, input_string, inputStringLen, 0);
  if (count != inputStringLen) {
    error("send() sent a different number of bytes than expected");
  }
  close(sock);
}
```

Die so erstellte Basisfunktionalität war nötig, damit zukünftige Funktionen via Kommandozeile einfach getestet werden konnten.

4.2 Verarbeitung der Befehle

Als nächster Schritt wurden die Editorbefehle implementiert. Gemäss Aufgabenstellung sollte der Server folgende fünf Befehle unterstützen:

- InsertLines: fügt Zeilen in das Dokument ein
- ReplaceLines: ersetzt Zeilen im Dokument
- ReadLines: gibt den Inhalt von Zeilen aus
- DeleteLines: löscht ganze Zeilen aus dem Dokument
- NumLines: gibt die Anzahl an Zeilen aus

Der besseren Übersichtlichkeit zuliebe wurde die Verarbeitung der Befehle in ein externes File *editorFunctions.h* ausgelagert. Dieses wurde in der Hauptdatei via *include* angezogen.

Mit folgendem Codeausschnitt wurden die Befehle verarbeitet:

```
void runCommand(char inputArg[]) {
  if (strcmp(inputArg,"InsertLines") == 0) {
    InsertLines();
  } else if (strcmp(inputArg,"ReplceLines") == 0) {
    ReplaceLines();
  } else if (strcmp(inputArg,"ReadLines") == 0) {
    ReadLines();
  } else if (strcmp(inputArg,"DeleteLines") == 0) {
    DeleteLines();
  } else if (strcmp(inputArg,"NumLines") == 0) {
    NumLines();
  } else {
    printf("Invalid command\n");
  }
}
```

Im Serverteil wurde daher folgender Codeausschnitt ersetzt:

Vorher:

```
printf("Client wrote: %s\n", buffer);
```

Nachher:

```
runCommand(buffer);
```

Statt, dass die vom Client empfangenen Texte auf der Kommandozeile ausgegeben wurden, ermöglichte der Aufruf der Funktion *runCommand* einen Textvergleich – wurde ein valider Befehl geschickt, wird die entsprechende Funktion aufgerufen.

Die Editor-Funktionen waren zu diesem Zeitpunkt noch von wenig Hilfe. Hier am Beispiel der Funktion *InsertLines*:

```
void InsertLines() {
  printf("Starting function InsertLines\n");
}
```

4.3 Bei Verbindung neuer Prozess

Eine Knacknuss im Projekt war das Prozesshandling bei neuen Client-Verbindungen. Bisher wurde pro Serverprozess lediglich eine Verbindung erwartet – mehrere Verbindungen waren nicht möglich.

Wie im Kapitel 3.1 konzeptioniert, sollte bei einem Verbindungsaufbau jeweils ein eigener Prozess für den Client und seine Befehle zur Verfügung stehen. Bisher jedoch war dies nicht

möglich, da der Verbindungsaufbau einmalig vor der Ausführung des Loops geschieht. Also musste der Server insofern umgestaltet werden, dass innerhalb des Loops auf den Verbindungsaufbau gewartet und der Mitteilungsempfang gänzlich in den Child-Prozess ausgelagert wird.

Zu diesem Zweck wurde der Loop wie folgt umgeschrieben:

```
while(1) {
  int clientSock = accept(sock, (struct sockaddr *) & clientAddress,
    &clientAddressLen);
  childpid = fork();
  if (childpid == -1) {
    perror("Fork failed!");
  }
  if (childpid == 0) {
    runProcess(clientSock);
    break;
  }
}
```

Die Funktion *accept* wartet dabei solange, bis ein Verbindungsaufbau eingeleitet wird. Danach wird mittels *fork* ein neuer Prozess gestartet. Im neuen Prozess wird dabei die Funktion *runProcess* ausgeführt, während im Serverprozess bereits auf den nächsten Verbindungsaufbau gewartet wird.

Die Funktion *runProcess* ist dabei für den Mitteilungsempfang verantwortlich:

```
while (1) {
  char buffer[256];
  bzero(buffer,256);
  read(clientSock,buffer,255);
  if (strcmp(buffer,"EXIT") == 0) {
    printf("Client disconnected\n");
  break;
  } else {
    runCommand(buffer);
  }
}
```

Auch hier befindet sich wieder ein endloser Loop, welcher auf neue Mitteilungen seitens Client wartet. Dabei handelt es sich grundsätzlich um den gleichen Loop, welcher in Kapitel 4.1 erklärt wurde. Lediglich die Abbruchvariable *breakUp* wurde durch das Keyword *break* ersetzt.

Durch diese Anpassung war es ab sofort möglich, mehr als einen Client gleichzeitig zu bedienen. Pro Client wird so ein eigener Kind-Prozess gestartet, welcher mit dem Empfang und der Verarbeitung der Befehle beauftragt ist.

4.4 Multi-Line Befehle

Die bisherige Befehlsverarbeitung war simpel: im empfangenen String wird nach einer passenden Zeichenkette – beispielsweise *ReplaceLines* – gesucht und bei Übereinstimmung die entsprechende Funktion aufgerufen. Gemäss Programmanforderung war es jedoch nötig, nebst dem eigentlich Befehlsaufruf noch weitere Parameter mitgeben zu können. Bei

der Funktion *ReplaceLines* müssen zum Beispiel die zu ersetzenden Zeilen angegeben werden können. Ausserdem muss natürlich der neue Zeilentext spezifiziert werden.

Gemäss Anforderung könnte ein solcher Aufruf beispielsweise so aussehen:
Client sendet:

```
ReplaceLines 5 3
Lorem ipsum dolor sit amet, consetetur sadipscing elitr
sed diam nonumy eirmod tempor invidunt ut labore et
dolore magna aliquyam erat, sed diam voluptua.
```

Dieser Aufruf ersetzt den Inhalt der Zeilen 5 bis 7 durch den angegebenen Content. Es handelt sich bei den empfangenen Befehlen also um mehrzeilige Eingaben. Bisher erwartet der Server jedoch bei jedem Aufruf zuerst den Funktionsnamen, also musste eine Möglichkeit geschaffen werden, um mehrzeilige Befehle empfangen und verarbeitet werden können.

Um dies zu ermöglichen, wurde ein Integer als Schalter implementiert, in welchem gespeichert werden kann, ob sich der Prozess in der Ausführung eines Befehls befindet und falls ja, in welchem Befehl er gerade ist. Bei einer neuen Client-Verbindung steht dieser Integer *isInCommand* auf dem Wert null – dies bedeutet, der Prozess befindet sich nicht in der Befehlsausführung sondern wartet auf einen Befehlsaufruf.

Der Client sendet:

```
InsertLines 10 1
```

Der Prozess empfängt dies und startet die Funktion *InsertLines*. Ausserdem setzt er den Schalter *isInCommand* auf den Wert eins – wobei eins hierbei dem Befehl *InsertLines* entspricht.

Der Client sendet:

```
Inhalt der neuen Zeile 10
```

Der Prozess empfängt dies und weiss nun auf Grund des vorher gesetzten Schalters, dass es sich hierbei nicht um einen Befehlsaufruf handelt sondern um zusätzlich Argumente zum vorher aufgerufenen Befehl *InsertLines*. Er führt den Befehl mit Hilfe dieser Argumente aus.

Ist der Befehl abgeschlossen – beispielsweise, da alle gewünschten Zeilen eingefügt wurden – setzt der Prozess den Schalter *isInCommand* wieder zurück auf null. Beim nächsten Input vom Client wird also wieder ein Befehlsaufruf erwartet.

4.5 Concurrency - Lockingfunktionalität

Als grosser Schwerpunkt der Seminararbeit sind die Lockingfunktionalitäten zu betrachten. Diese sollen gemäss Konzept – siehe Kapitel 3.3 – aus einem Array im Shared Memory mit Absicherung via Semaphore bestehen.

Im aktuellen Versionsstand des Servers sind bereits mehrere Client-Verbindungen möglich. Dies führt jedoch unweigerlich zu einem Konflikt: was passiert, wenn zwei Clients

gleichzeitig auf eine Zeile zugreifen möchten? Um dieses Concurrency-Problem zu lösen, müssen die genannten Lockingfunktionalitäten konstruiert und anschliessend implementiert werden.

Um die Abstraktion des Codes zu bewahren, wurden für die ganzen Lockingmechanismen ein eigenes Header-File *lockingFunctions.h* erstellt. Die folgenden Funktionen sind in diesem File abgelegt.

4.5.1 Shared Memory initialisieren

In einem ersten Schritt wurden die Vorbereitungen für die Verwendung von Shared Memory implementiert. Zu diesem Zweck wurde eine Funktion *setup_shm* erstellt, welche das Shared Memory vorbereitet. Diese Funktion wird jeweils beim Serverstart noch vor dem Loop ausgeführt:

```
int setup_shm() {
  shmid = shmget(key_sh, 1024, IPC_CREAT | 0666);
  if(shmid < 0) {
    error("Error in shmget");
    return 1;
  } else {
    lockedLines = (int *)shmat(shmid, 0, 0);
    return 0;
  }
}
```

Wird diese Funktion beim Serverstart ausgeführt, reserviert sie ein entsprechender Bereich im Shared Memory und legt darin das Integer-Array *lockedLines* ab. Bereits jetzt kann auf dieses Array prozessübergreifend zugegriffen werden.

4.5.2 Lockingfunktionen

Um das Array sinnvoll verwenden zu können, wurden natürlich entsprechende Funktionen bereitgestellt. Dabei hatte sich gezeigt, dass die folgenden müssen:

Eine Zeile sperren:

```
void lockLine(int lineNum) {
  lockedLines[lineNum] = 1;
}
```

Mehrere Zeilen sperren:

```
void lockMultipleLines(int startLine, int endLine) {
  int i;
  for (i = startLine; i <= endLine; i++) {
    lockLine(i);
  }
}
```

Eine Zeile entsperren:

```
void unlockLine(int lineNum) {
  lockedLines[lineNum] = 0;
}
```

Mehrere Zeilen entsperren:

```
void unlockMultipleLines(int startLine, int endLine) {
  int i;
  for (i = startLine; i <= endLine; i++) {
    unlockLine(i);
  }
}
```

Prüfen, ob eine Zeile gesperrt ist:

```
int isLineLocked(int lineNum) {
  return lockedLines[lineNum];
}
```

Prüfen, ob mindestens eine von mehreren Zeilen gesperrt ist:

```
int areLinesLocked(int startLine, int endLine) {
  int i;
  int result = 0;
  for (i = startLine; i <= endLine; i++) {
    if (isLineLocked(i) == 1) {
      result = 1;
    }
  }
  return result;
}
```

4.5.3 Server anpassen

Um sinnvollen Gebrauch dieser Funktionen zu machen, muss natürlich der Server entsprechend angepasst werden. Insbesondere musste der Server neu die entsprechenden Abfragen vor der Ausführung eines Befehls tätigen, um zu prüfen, ob Zeilen gesperrt sind oder ein Zugriff möglich ist. Wichtig war auch, dass im Falle einer gesperrten Zeile die Befehlsausführung nicht gänzlich abgebrochen, sondern lediglich ‚on hold' gesetzt wird. Der Client darf somit in jedem Fall von einer Befehlsausführung ausgehen und muss diese nicht erneut anstossen.

Die Implementation soll beispielhaft an Hand des Befehls *ReadLines* dargestellt werden. Bei Ausführung dieses Befehls sollen folgende Schritte durchlaufen werden:

1. Befehl wird aufgerufen
2. Prüfen, ob eine der zu lesenden Zeilen gesperrt ist
3. Wenn ja, eine Sekunde warten, sonst weiterfahren
4. Erste zu lesende Zeile sperren
5. Zeile lesen
6. Zeile freigeben
7. Nächste zu lesende Zeile sperren
8. Zeile lesen
9. Zeile freigeben
10. Inhalt der Zeilen an den Client senden
11. Befehlsausführung verlassen

Verglichen damit der eigentliche Code[2]:

```
while (areLinesLocked(startNum, endLine) == 1) {
  sleep(1);
}
for (i = startNum; i <= endLine; i++) {
  lockLine(i);
  // here comes the file modification
  printf("Read line %i\n", i);
  char content[256];
  memset(content, 0, sizeof(content));
  sprintf(content, "\nContent of line %i", i);
  strcat(response, content);
  unlockLine(i);
}
printf("\n");
printToClient(response);
```

4.6 Signalhandler für Freigabe von Socket und SHM

Oftmals trat beim Neustart des Servers die folgende Fehlermeldung auf:

```
Bind failed. Error: Address already in use.
```

Dies lag daran, dass bei einem unsachgemässen Beenden des Servers – beispielsweise durch einen Absturz oder der Tasteneingabe CTRL + C – der Socket nicht korrekt geschlossen wurde. Zwar war dies unschön, doch bisher stellte das Verhalten kein grosses Problem dar – musste doch lediglich kurz gewartet werden, dass sich der Socket von selber schliesst.

Mit der Einführung von Shared Memory im Kapitel 4.5 wurden unsachgemässe Abbrüche jedoch zum Problem. Würde der Server abstürzen, war jetzt neu neben dem noch offenen Socket auch das Shared Memory nicht korrekt aufgeräumt und würde beim erneuten Start zu Fehlfunktionen führen. Also musste eine entsprechende Lösung implementiert werden.

Zu diesem Zweck wurde ein Signalhandler eingeführt, welcher beim Empfang des Signals *SIGINT* – beispielsweise bei Eingabe von CTRL + C – die Funktion *shutdownServer* ausführt. Diese Funktion wiederum beendet Client-Verbindungen, schliesst noch offene Sockets und räumt das Shared Memory auf.

4.7 Ausgabe an Client statt Kommandozeile

Bisher wurden alle Ausgaben lediglich auf der Kommandozeile des Server beziehungsweise des entsprechenden Prozesses ausgegeben. Der Client selber hat keine Bestätigung darüber erhalten, ob der Befehl korrekt ausgeführt wurde. Natürlich ist dies wenig sinnvoll – die Ausgabe muss an den Client übertragen werden.

Um dies einfach und übersichtlich zu halten, wurde eine neue Funktion *printToClient* konzeptioniert. Diese sollte anaolog der Funktion *printf* funktionieren, jedoch statt auf der Kommandozeile, Text direkt an den Client senden.

[2] Auszug Zeile 126 -140 aus editorFunctions.h

Die Funktion sieht wie folgt aus:

```
void printToClient(char stringToSend[]) {
  char *sendText = stringToSend;
  send(clientSock, sendText, strlen(sendText), 0);
}
```

Diese Funktion konnte jetzt überall dort verwendet werden, wo bisher mittels *printf* Text auf der Kommandozeile ausgegeben wurde. Als Beispiel dient hier ein Ausschnitt aus dem Befehl *DeleteLines.* Gemäss Spezifikation soll jeweils nach Abschluss des Löschvorgangs der Text *DELETED* an den Client übertragen werden. Somit war folgende Anpassung nötig:

Vorher:

```
printf("DELETED");
```

Nachher:

```
printToClient("DELETED");
```

4.8 Automatisches Testing

Bisher diente der Test-Client dazu, manuelle Eingaben an den Server zu senden und dessen Antwort auf der Kommandozeile auszugeben. Diese Funktionalität sollte zwar auch weiterhin verfügbar sein, jedoch wäre es wesentlich einfacher, wenn beim Aufruf des Test-Clients jeweils alle Befehle automatisch durchlaufen würden – der User muss diese nicht mehr händisch eingeben.

Zu diesem Zweck wurde der Testclient *./test* durch eine Argumentenabfrage erweitert. Neu sollte beim Aufruf via

```
./test
```

das automatische Testing beginnen, während beim Aufruf von

```
./test –manual
```

die bereits bekannte Funktionalität der manuellen Eingabenverarbeitung gestartet werden soll.

5 Verwendung

Die vorliegende Software wurde in einer UNIX Umgebung – genauer Ubuntu 12.04 mit POSIX Konfiguration – entwickelt und getestet. Obwohl die Software grundsätzlich plattformunabhängig ist, kann eine einwandfreie Funktionalität kann nur auf einem UNIX System garantiert werden. Eine weitere Voraussetzung für den fehlerfreien Betrieb ist ein aktueller C-Compiler, zum Beispiel *gcc*[2].

Das Projekt einhält ein Makefile. Dies bedeutet, sobald das GitHub Repository[3] lokal geklont wurde, kann mittels folgendem Befehl die Kompilierung des Programms gestartet werden:

```
make
```

Dabei muss sich der Benutzer im Wurzelverzeichnis der Software befinden. Nach Abschluss der Kompilierung sind die folgenden ausführbaren Dateien verfügbar:

- run
- test

5.1 Der Serverteil ‚run'

Als erstes soll der Server via Kommandozeile gestartet werden:

```
./run
```

Via Ausgabe wird der Benutzer darüber informiert, dass der Server erfolgreich gestartet wurde und auf eine eingehende Verbindung gewartet wird.

Der Server hört dabei auf dem fest vorgegebenen Port 2354.

5.2 Der Test-Client ‚test'

Natürlich kann jeder beliebige TCP-Client verwendet werden um mit dem Server zu kommunizieren. Auch ein Client, geschrieben in einer anderen Programmiersprache wäre denkbar. Der im Rahmen des Projekts erstellte Test-Client besitzt nur eine rudimentäre Oberfläche und limitierte Funktionalität – erfüllt jedoch seinen Zweck.

Der Test-Client kann – wie bereits unter Punkt 4.8 erklärt – in zwei Modi ausgeführt werden. Beim Aufruf via Kommandozeile von

```
./test
```

wird das automatische Testing gestartet, während beim Aufruf via

```
./test –manual
```

die manuellen Eingabenverarbeitung gestartet wird.
Im automatischen Modus werden alle verfügbaren Befehle in folgender Reihenfolge mit folgenden Parametern durchgetestet:

- InsertLines 1 2
- ReplaceLines 5 2
- ReadLines 1 10
- DeleteLines 10 10
- NumLines

Wird jedoch die händische Eingabenverarbeitung mittels des Switchs –manual ausgewählt, kann der Benutzer auf der Kommandozeile Befehle von Hand eingeben. Nach dem Bestätigen mittels Enter-Taste wird der String an den Server geschickt.

Der Test-Client sendet Verbindungsanfragen an die IP-Adresse 127.0.0.1 über den Port 2354.

6 Beschränkungen

Im Laufe der Umsetzung traten verschiedenste Probleme und Fehler auf. Die meisten davon konnten glücklicherweise im Rahmen des zur Verfügung stehenden Zeitbudgets korrigiert werden. Trotzdem gibt es auch in der aktuellen Version noch gewisse Beschränkungen. Die wichtigste davon wurde unter Kapitel 1.2 bereits erklärt und wird hier nicht nochmal aufgeführt.

6.1 Shared Memory und Semaphore

Trotz intensiver Recherche war es mir nicht möglich, das Shared Memory mittels Semaphore abzusichern. Im produktiven Einsatz wäre dies denkbar ungünstig, denn zwar sind Lockingmechanismen für einen gleichzeitigen Serverzugriff implementiert, aber können durch den Einsatz von ungesichertem Shared Memory Race Conditions auftreten.

Ein Beispiel – ausgehend von einem Server-Neustart:

1. Client A liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist frei.
2. Client A sperrt Zeile 6.
3. Client B liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist gesperrt.
4. Client B wartet.
5. Client A tätigt Filemodifikationen.

Somit hat alles seine Richtigkeit. Wenn jetzt aber Client B zwischen Schritt 1 und Schritt 2 die Abfrage tätigt, kann es zu Race Conditions kommen:

1. Client A liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist frei.
2. Client B liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist frei.
3. Client A sperrt Zeile 6.
4. Client B sperrt Zeile 6.
5. Client A tätigt Filemodifikationen.
6. Client B tätigt Filemodifikationen.

Mittels Semaphore kann das Shared Memory insofern abgesichert werden, dass nur ein Zugriff möglich ist, wenn vorher die Semaphore erworben werden konnte.

Das obige Beispiel würde sich dann wie folgt abspielen:

1. Client A erwirbt die Semaphore.
2. Client A liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist frei.
3. Client B versucht, die Semaphore zu erwerben – dies ist nicht möglich, darum wartet Client B auf die Freigabe der Semaphore
4. Client A sperrt Zeile 6.
5. Client A gibt die Semaphore frei.
6. Client B erwirbt die Semaphore.
7. Client B liest im Shared Memory, ob die Zeile 6 gesperrt ist: Zeile 6 ist gesperrt.
8. Client B wartet.
9. Client A tätigt Filemodifikationen.

6.2 Statische Memory Allokation

Eine weitere Beschränkung ist die statische Allokation von Memory. Für Strings wird jeweils immer nur ein vorher fix definierter Speicher in der Grösse von 1024 Zeichen zur Verfügung gestellt. Dies bedeutet, die Befehle sowie deren Argumente können maximal eine Länge von 1024 Zeichen besitzen.

Auch die vom Server an den Client zurückgesendeten Informationen können maximal 1024 Zeichen lang sein. Sollte also ein Dokument bearbeitet werden, welches Zeilen besitzt, die länger als 1024 Zeichen sind, müsste die Applikation angepasst werden.

6.3 Server-Response im Test-Client

Sollte der Test-Client im manuellen Modus ausgeführt werden, so erwartet dieser nach jeder Befehlseingabe eine Rückmeldung vom Server.

Dies kann insofern zu Problemen führen, als das der Server beispielsweise beim Aufruf der Funktion *InsertLines* erst nach Abschluss – also Einfügen der letzten Zeile – die Antwort *INSERTED* an den Klienten schickt.

Dies ist in der aktuellen Version des Test-Clients ein Fehler und muss berücksichtigt werden. Der automatische Modus hat diesen Fehler nicht.

7 Fazit

7.1 Rückblick

Obwohl ich im Rahmen dieser Seminararbeit mit viel Frust gekämpft habe und mich teilweise die Umsetzung richtig Nerven gekostet hat, so muss ich gestehen, dass mich das Resultat mit Stolz erfüllt.

Am Kick-Off fand ich Interesse am Thema *Concurrency*, aber bereits da wusste ich, dass die Umsetzung mit viel Recherche und harter Arbeit verbunden ist. Im Laufe des Seminars

musste ich feststellen, dass meine Kenntnisse in der Programmiersprache C nicht ansatzweise ausreichten, um dieses Projekt zu verwirklichen – ich kam nur sehr mühselig voran.

Glücklicherweise blieb ich dran und kämpfte mich vorwärts. Obwohl das Zeitbudget um Längen überschritten und gewisse Punkte noch immer nicht umgesetzt sind, kann ich diese Seminararbeit abschliessen und bin stolz darauf.

Das Thema *Concurrency* habe ich begriffen und sogar selbstständig umgesetzt.

7.2 Lessons learned

Zwei wichtige Punkte kann ich für mich mitnehmen und werde auch in Zukunft daran festhalten.

Zum einen ist es wichtig, dass man nicht aufgibt sondern stets weiter arbeitet. Ohne dies hätte ich diese Seminararbeit nicht erfolgreich abschliessen können.

Zum anderen weiss ich es jetzt sehr zu schätzen, welche Arbeit mir eine objektorientierte Programmiersprache, wie beispielsweise Java abnimmt.

7.3 Dank

Ich bedanke mich an dieser Stelle bei allen Personen, die mich bei dieser Seminararbeit unterstützt haben. Insbesondere bei meinen Kommilitonen und bei Delia fürs Korrekturlesen.

8 Quellenangabe / Literaturverzeichnis

[1] https://github.com/telmich/zhaw_seminar_concurrent_c_programming

[2] http://gcc.gnu.org/

[3] https://github.com/Schnabulation/Concurrent_C