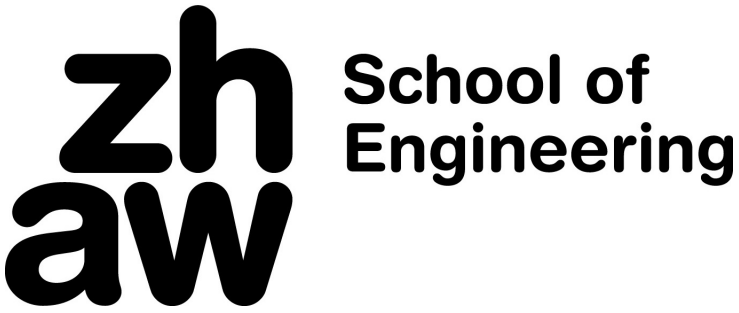
Zürcher Hochschule für Angewandte Wissenschaften

Seminararbeit

Concurrent C Programming



Dozent:
Nico Schottelius
xsou@zhaw.ch

Kandidat:
Benjamin Bütikofer
bbutik@microsoft.com
Matr.: 11-480-993

19. Juni 2015

Inhaltsverzeichnis

1 Einleitung

1.1 Abstract

Dieser Bericht beschreibt die Implementierung eines multiuser und netzwerkfähigen Client/Server-Spiels welches in C geschrieben wurde. Die Arbeit wurde im Seminar "Concurrent C Programming" durchgeführt. Das Seminar wurde im Rahmen der Kurse Systemsoftware und Betriebssystemprogrammierung besucht.

1.2 Spielziel

Ziel des Spiels ist es alle Felder des Spielfelds zu besitzen. Das Spielfeld ist ein Quadrat der Seitenlänge n, wobei

$$n \geq 4$$

ist. Die Koordinaten des Spielfeldes sind somit

$$(0..(n-1), 0..(n-1))$$

.

1.3 Spielablauf

1. Der Server startet und wartet auf $\frac{n}{2}$ Spieler.
2. Sobald $\frac{n}{2}$ Spieler verbunden sind, kann jeder Spieler versuchen Felder zu erobern.

3. Es können während des Spiels neue Spieler hinzukommen oder Spieler das Spiel verlassen.
4. Der Server prüft alle y Sekunden den konsistenten Spielfeldstatus , wobei

$$1 \leq y \leq 30$$

   gilt.
5. Wenn ein Spieler zum Zeitpunkt der Prüfung alle Felder besitzt, hat er gewonnen und das Spiel wird beendet.

1.4 Spiel-Protokol

- Befehle werden mit \n abgeschlossen.
- Kein Befehl ist länger als 256 Zeichen inklusive dem \n.
- Jeder Spieler kann nur ein Kommando senden und muss auf die Antwort warten.

1.4.1 Anmeldung

Eine Anmeldung kann erfolgreich order nicht erfolgreich sein.

```
Client: HELLO\n
Server: SIZE n\n
```

Listing 1.1: Erfolgreiche Anmeldung

```
Client: HELLO\n
Server: NACK\n
    -> Trennt die Verbindung
```

Listing 1.2: Nicht erfolgreiche Anmeldung

1.4.2 Spielstart

Sobald $\frac{n}{2}$ Verbindungen hergestellt sind, schickt der Server das START Signal an alle Teilnehmer.

```
Server: START\n
Client: - (erwidert nichts, weiss das es gestartet hat)
```

Listing 1.3: Spielstart

1.4.3 Feld erobern

Wenn kein anderer Client gerade einen TAKE Befehl für das selbe Feld sendet, kann ein Client es nehmen.

```
Client: TAKE X Y NAME\n
Server: TAKEN\n
```

Listing 1.4: Erfolgreiche Eroberung

Wenn ein oder mehrere andere Clients gerade einen TAKE Befehl für das selbe Feld sendet, sind alle bis auf den ersten nicht erfolgreich.

```
Client: TAKE X Y NAME\n
Server: INUSE\n
```

Listing 1.5: Nichterfolgreiche Eroberung

1.4.4 Besitz anzeigen

```
Client: STATUS X Y\n
Server: Name-des-Spielers\n
```

Listing 1.6: Besitz anzeigen

1.4.5 Spielende

Sobald ein Client alle Felder besitzt wird der Gewinner bekanntgegeben. Diese Antwort kann auf jeden Client Befehl, mit Ausnahme der Anmeldung, kommen.

```
Server: END Name-des-Spielers\n
Client: - (beendet sich)
```

Listing 1.7: Spielende

1.5 Bedingungen für die Implementation

- Es gibt keinen globalen Lock.
- Der Server speichert den Namen des Feldbesitzers.
- Kommunikation via TCP/IP.
- fork + shm (empfohlen).
  - oder pthreads.
  - für jede Verbindung einen Prozess/Thread.
  - Hauptthread/prozess kann bind/listen/accept machen.
  - Rating Prozess/Thread zusätzlich im Server.
- Fokus liegt auf dem Serverteil.
- Client ist hauptsächlich zum Testen und “Spass haben” da.
- Server wird durch Skript vom Dozent getestet.
- Locking, gleichzeitiger Zugriff im Server lösen.
- Debug-Ausgaben von Client/Server auf stderr.

2 Planung und Implementation

Bevor mit der Implementation begonnen wurde, wurden die Anforderungen analysiert und der Spielablauf mit einem simplen Blocks-and-Arrows Diagram zu Papier gebracht.

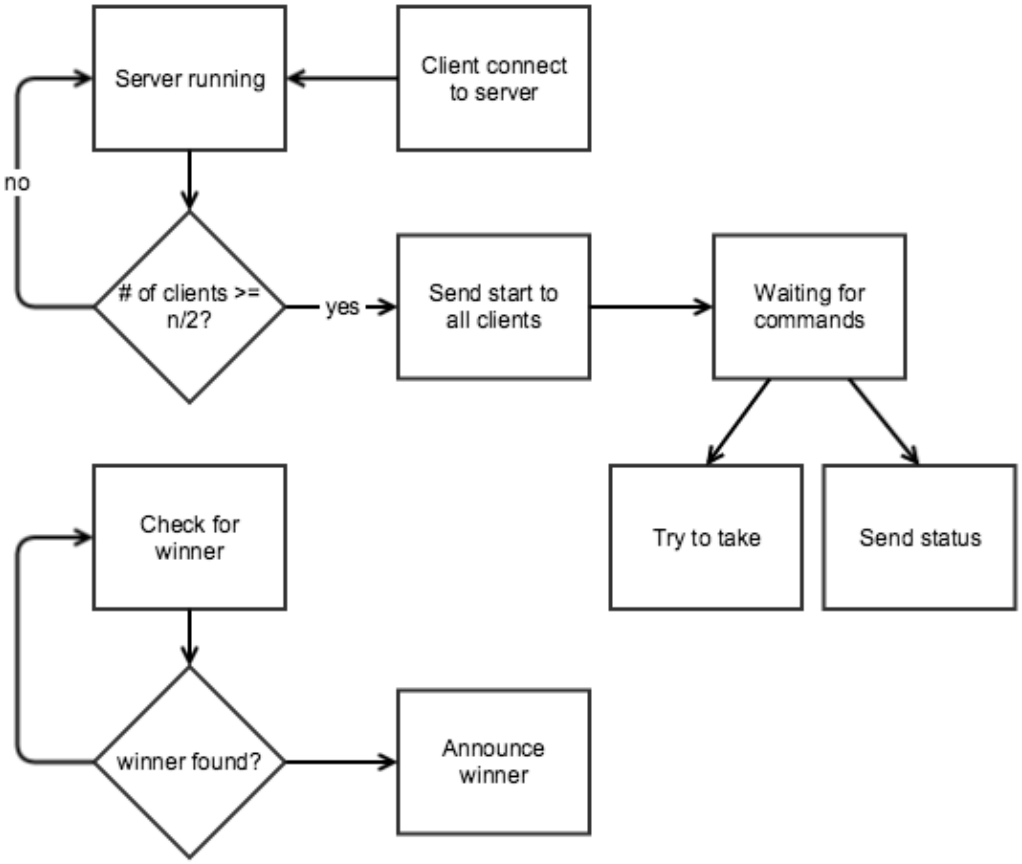


Abbildung 2.1: Spielablauf

Mit dieser Visualisierung konnte im nächsten Schritt die Anforderungen an den Server ermittelt werden. Danach wurde ein weiteres Diagram angefertigt, welches die Funktionalität des Servers und die Abhängigkeiten der einzelnen Komponenten des Servers aufzeigt, siehe Abbildung 2.2.

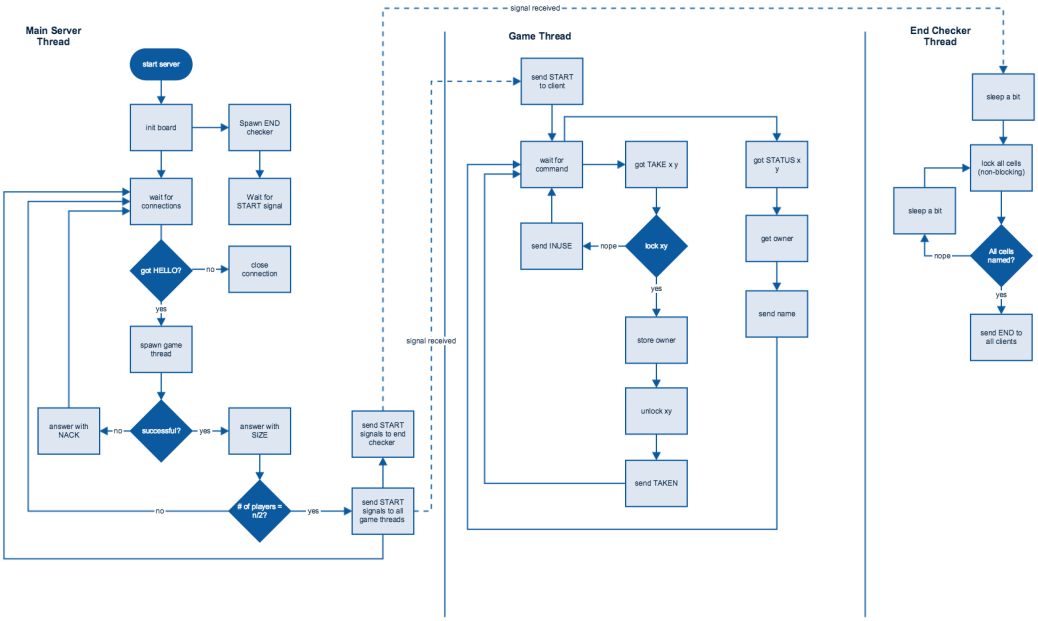


Abbildung 2.2: Blocks and Arrows Diagram Server Prozess. Das Diagram ist in voller Auflösung auch im Anhang zu dieser Arbeit zu finden.

Das Erstellen dieses Diagrams beanspruchte einiges an Zeit und war ein iterativer Prozess, weil nach jeder Version neue Fallstricke ersichtlich wurden. Doch die Arbeit lohnte sich, da die Grafik die Implementation enorm vereinfachte weil klar ersichtlich wurde, was welche Komponente zu erfüllen hat.

2.1 Server

Wie Abbildung 2.2 zu entnehmen ist, besteht der Server aus drei Hauptkomponenten:

- Main Server Thread
- Game Thread
- End Checker Thread

2.1.1 Main Server Thread

Der Main Server Thread ist verantwortlich den Server zu starten und neue Verbindungen anzunehmen und zu speichern. Sobald genügend Clients verbunden

sind, soll er auch die START Nachricht wie in 1.4.2 definiert an alle Clients versenden.

Beim Starten des Servers wird als erstes die Umgebung aufgesetzt. Das beinhaltet neben dem Registrieren der Signal-Handlern auch das initialisieren des Logging Frameworks, dem Verarbeiten der Kommandozeilenparametern und dem Setup des Spielfeldes. Um allen Clients den Start-Befehl zur selben Zeit zu schicken, wird eine Pthread-Barriere verwendet und beim Programstart initialisiert.

Sind all diese Schritte erfolgreich abgeschlossen, wird in einem neuen Thread der Connection Handler gestartet welcher, wie der Name schon sagt, die eingehenden Verbindungen verarbeiten wird.

2.1.1.1 Connection Handler

Der Connection Handler basiert auf den Erklärungen von [1] und [3]. Wenn eine neue Verbindung zum Server erstellt wird, wird dieser ein neuer File Descriptor zugewiesen. Danach wird ein neues Spieler struct erstellt und der File Descriptor wird darin gespeichert. Nun ist die Verbindungsinitialisierung komplett und ein Game Thread wird erstellt. Falls etwas schiefgeht wird ein NACK an den Client geschickt und die Verbindung wird terminiert.

2.1.1.2 Verwendete Datenstrukturen

Die Spielinformationen werden in drei Structs verwaltet. Kernighan und Ritchie definieren ein `struct` als

> (...) an object consisting of a sequence of named members of various types.

[2]

Die Spielerinformation wird in `player_t` gespeichert und beinhaltet neben dem Namen auch den File Descriptor.

```
1 typedef struct {
2     char *name;
3     int id;
4     int fd;
```

```
5     pthread_t tid;
6     int num_of_cells;
7 } player_t;
```

Listing 2.1: Player Struct

Ein Spielfeld (Board) besteht aus mehreren Zellen. Da eine Zelle mehrere Informationen verwalten muss, gibt es auch für die Zelle ein eigenes Struct. Im Board stehen neben einem pointer zu einer Zelle auch noch weitere Informationen wie zum Beispiel die Grösse des Spielfeldes und die Anzahl der angemeldeten Spieler.

```
1 typedef struct {
2     int n;
3     int num_players;
4     pthread_t server_tid;
5     pthread_t checker_tid;
6     _Bool game_in_progress;
7     cell_t *cells;
8 } board_t;
```

Listing 2.2: Board Struct

Die Zelle ist sehr einfach gehalten. Jede Zelle hat einen Mutex und einen Pointer zu einem Spieler. Ist das Feld nicht vergeben, wird der dummy Spieler “free” gespeichert.

Der Mutex wird verwendet, um zu verhindern, dass zwei Spieler gleichzeitig eine Zelle in Besitz nehmen können. Will also ein Spieler eine Feld annektieren, wird zu erst versucht den Lock zu erhalten. Ist dieser Schritt erfolgreich, wird der Name des Spielers in das Feld geschrieben, ansonsten wird die Meldung `INUSE` an den Spieler geschickt.

```
1 typedef struct {
2     pthread_mutex_t cell_mutex;
3     player_t *player;
4 } cell_t;
```

Listing 2.3: Cell Struct

2.1.2 Game Thread

Der Game Thread ist eine unendliche `while`-Schleife die mit `recv` auf einkommende Befehle hört.

Wenn eine Nachricht Empfangen wird, werden zuerst allfällige Steuerzeichen (\n, \r) aus dem Buffer entfernt und danach die Daten geparsed, verarbeitet und die entsprechende Antwort an den Client geschickt.

Handelt es sich dabei um ein `HELLO`, wird geprüft wieviele Spieler am Server angemeldet haben. Trifft der Fall

$$(Anzahl\ Spieler) \geq \frac{n}{2}$$

ein, wird die Barriere aufgehoben und alle Spieler erhalten zur gleichen Zeit das `START` Kommando. Zusätzlich wird der End Checker Thread gestartet. Falls nicht genügend Spieler online sind, wartet der Thread an der Barriere bis sie aufgehoben wird. Ist das Spiel schon gestartet, wird sofort der Start Befehl ausgelöst.

2.1.3 End-Checker

Der End-Checker Thread läuft wie der Connection Handler in einer Dauerschleife bis ein Gewinner ermittelt ist. Git es einen Gewinner, schickt der End-Checker die END Nachricht.

In der Aufgabenstellung wird gefordert, dass kein globaler Lock verwendet wird. Es wird aber auch verlangt, dass das Spielfeld konsistent geprüft werden muss. Da sich diese zwei Anforderungen jedoch nicht vereinbaren lassen, habe ich mich entschieden eine Lösung ohne globalen Lock zu entwickeln. Dies hat jedoch zur Folge, dass ein Spielfeld welches am Anfang getestet wird beim Abschluss der Prüfung aller Felder einen neuen Besitzer haben kann.

Wenn das Check-Timeout abgelaufen ist, iteriert der End-Checker durch das gesamte Board und versucht die Zelle mit einem Mutex zu locken. Dabei wird ein locking-mutex verwendet, der Checker bleibt also bei jeder Zelle so lange stehen, bis er den Lock erfolgreich erhalten hat.

Sobald der Lock erfolgreich akquiriert wurde, wird geprüft, ob der momentane Zellen Inhaber derselbe ist, wie von der Zelle vorher. Sind beide Besitzer gleich,

wird der Lock abgegben und mit der nächsten Zelle fortgefahren, falls nicht, wird die Überprüfung angehalten, der Lock freigegeben und wieder eine zufällige Zeit gewartet.

Falls der Checker einen Gewinner ermittelt hat, wird jedem Spieler die `END` Nachricht geschickt und der Thread beendet sich.

2.2 Client

Da die Zeit etwas knapp wurde, ist nur eine Client Strategie implementiert worden.

2.2.1 Retry once

Der Client beginnt beim Feld mit den Koordinaten $(1, 1)$ und versucht das Feld zu besetzen. Ist der Versuch das Feld zu belegen gescheitert, wird ein weiteres Mal versucht das Feld in besitz zunehmen. Schlägt auch der zweite Versuch fehl, wird mit dem nächsten Feld weitergemacht.

Weitere Optimierungen sind wegen Zeitmangels nicht implementiert worden. Folgende weiteren Algorithmen wurden angedacht:

- Zufall: Die Reihenfolge der Felder die besetzt werden sollen, wird rein zufällig gewählt.
- Reverse: Anstatt von vorne zu beginnen, wird von (n, n) gestartet.
- Divide and Conquer: Das Spielfeld wird in der Mitte geteilt und zwei Threads wenden einer der oben genannten Methoden an um die Felder zu besetzen.

3 Fazit

Das Projekt war äusserst interessant und hat neben viel Ärger auch sehr viel Freude bereitet. Es war das erste Projekt welches ich in C geschrieben habe, demzufolge war die Lernkurve auch ziemlich steil. Hinzukommt, dass ich beruflich selten programmiere und es schon eine Weile her war seit meinem letzten Programmier-Projekt.

Glücklicherweise ist beinahe jedes Problem schon einmal gelöst worden und es gibt ausreichend Dokumentation im Internet. Auch der Tipp mit valgrind war äusserst nützlich.

Das Projekt zeigte mir auch deutlich, warum Sprachen wie Java und C# soviel Auftrieb erhalten haben. Muss man sich doch in C jede noch so kleine Funktion in der Regel selbst bauen. Ein TCP Server in Java kann inner Minuten geschrieben werden ohne das man sich gross Gedanken machen muss, wie es im Hintergrund abläuft.

Auch der Umgang mit dem Speicher (Speicherreservierung) hat viel Nerven gekostet - umso schöner jedoch, wenn man dann endlich auch den letzten Fehler ausgeschliffen hat!

Abbildungsverzeichnis

Listings

Literatur

[1] URL: http://www.gnu.org/software/libc/manual/html_node/Server-Example.html.

[2] *The C Programming Language*. 2. Aufl. Prentice Hall, 1988.

[3] Stephen A. Rago W. Richard Stevens. *Advanced Programming in the Unix Environment*. 3. Aufl. Addison Wesley, 2013.