Seminar Systemsoftware

Concurrent C Programming

Dokumentation

Samuel Reutimann

Inhalt

0 Einleitung

0.1 Aufgabe / Projekt

Ein Forking / Multithreaded Client-Server Spiel erstellen.

0.2 Spielregeln

0.2.1 Ziel des Spiels

Eroberung aller Felder des Spielfeldes durch einen Spieler.

0.2.2 Spielaufbau

Das Spielfeld ist ein Quadrat der Seitenlänge n, wobei n >= 4 ist. Die Koordinaten des Spielfeldes sind somit (0..(n-1), 0..(n-1)).

0.2.3 Spielablauf

Der Server startet und wartet auf n/2 Spieler. Sobald n/2 Spieler verbunden sind, kann jeder Spieler versuchen Felder zu erobern. Es können während des Spiels neue Spieler hinzukommen oder Spieler das Spiel verlassen. Der Server prüft alle y Sekunden den konsistenten Spielfeldstatus, wobei 1 <= y <= 30. Wenn ein Spieler zu diesem Zeitpunkt alle Felder besitzt, hat er gewonnen und das Spiel wird beendet

0.3 Protokoll Allgemein

- Befehle werden mit \n abgeschlossen
- Kein Befehl ist länger als 256 Zeichen inklusive dem \n
- Jeder Spieler kann nur 1 Kommando senden und muss auf die Antwort warten

0.3.1 Anmeldung

Erfolgreiche Anmeldung:

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

Nicht erfolgreiche Anmeldung:

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

0.3.2 Spielstart

Der Server wartet auf n/2 Verbindungen vor dem Start.

```
Server: START\n
Client: -> erwidert nichts, weiss dass es gestartet hat
```

0.3.3 Feld erobern erfolgreich

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

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

0.3.4 Feld erobern: nicht erfolgreich

Wenn ein oder mehrere andere Clients gerade einen TAKE Befehl für dasselbe Feld sendet, sind alle bis auf der erste nicht erfolgreich.

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

0.3.5 Besitz anzeigen

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

0.3.6 Spielende

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

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

0.4 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

1 Anleitung zur Nutzung

Zur Nutzung des Programms wird ein Linux PC vorausgesetzt.

1.1 Downloaden

Download der Sourcefiles im /data Verzeichnis.

```
cd /data/

git clone
https://github.com/samfisch3r/zhaw_concurrent_c_programming_fs_2015.git
```

1.2 Kompilieren

```
make
```

Das make erstellt zwei Dateien:

- client
- server

1.3 Ausführung

1.3.1 Server

Zuerst muss der Server gestartet werden:

```
./server <size> [port]
```

Folgende Optionen sind notwendig/möglich:

Die <size> ist ein nötiger Parameter, welcher eine Zahl sein muss die mindestens 4 ist. Dadurch wird die Seitenlänge des Spielfeldes festgelegt. Dadurch wird ein Quadrat mit 2^<size> Feldern erstellt.

Der [port] Parameter ist freiwillig und kann verwendet werden um den Server auf einem spezifischen Port laufen zu lassen, der Standard Port ist hierbei 1234.

1.3.2 Client

Wenn der Server am Laufen ist, können sich die Clients daran anmelden. Die Clients werden folgendermassen gestartet:

```
./client <name> [port]
```

Der <name> Parameter ist notwendig, damit jeder Client sich mit einem eigenen alias beim Server melden kann.

Der [port] Parameter ist freiwillig und kann verwendet werden falls der Server auf einem alternativen Port läuft, der Standard Port ist hierbei 1234.

1.3.3 Spiel

Sobald sich genügend Spieler <size>/2 am Server angemeldet haben, beginnt das Spiel. Die Clients schicken nun Anfragen an den Server um ein Feld einzunehmen. Dies dauert so lange, bis ein Spieler alle Felder erobert hat. Passiert dies, schickt der Server den Namen des Gewinners an alle Clients. Danach beenden sich alle Clients und der Server.

2 Weg

2.1 Client/Server

Mein Ziel war es, zuerst einmal die Client/Server Infrastruktur aufzubauen. Dazu habe ich folgendes Guide [1] von Brian "Beej Jorgensen" Hall durchgearbeitet. Ich habe dies auch schon für frühere Projekte benutzt und wusste, dass es sehr informativ und auf einem aktuellen Stand ist. Da viele andere Guides zur Programmierung im C Netzwerkbereich schon etwas veraltet sind und sich somit die verschiedensten structs, welche benötigt werden noch von Hand auffüllen. Der Autor dieses Guides hat freundlicherweise auch gleich ein Beispiel für eine einfache Client/Server implementation auf GitHub veröffentlicht [2]. Ich habe diese implementation verwendet um meinen Client und Server darauf aufzubauen.

2.2 Spielfeld

Um die Möglichkeit zu haben mehrere Variablen pro Feld zu speichern, habe ich ein struct gewählt. Somit kann ich für jedes Feld den Namen des Besitzers speichern, sowie einen locking Mechanismus hinzufügen. Dies, da ein Globales Lock nicht erlaubt ist. Somit kann jedes Feld separat gelockt werden. Zur Ansteuerung der Felder habe ich ein zweidimensionales Array gewählt. Details zu dieser Struktur habe ich hier [3] gefunden. Im Verlaufe der Arbeit habe ich die Entscheidung ein zweidimensionales Array mit structs zu verwenden revidiert, da es einige Probleme mit dem locking und dem Shared Memory gegeben hatte. Somit setzte ich schlussendlich auf ein Eindimensionales Array. Ich konnte weiterhin mit x und y als Koordinaten arbeiten, da sich mit der einfachen Rechnung x+y*Seitenlänge ein Eindeutiger Platz im Array ergibt.

2.3 Fork oder Thread

Nachdem ich einen guten Artikel [4] gelesen habe zum Thema Fork/Thread, habe ich mich entschieden in meinem Programm Forks zu verwenden. Einige Positive punkte, welche mich zu dieser Entscheidung gebracht haben sind:

- Viel einfachere Entwicklung
- Der Code ist einfacher zu Warten
- Wenn ein Fork abstürzt, laufen das Programm und die anderen Forks weiter.

In der Entwicklung hat sich herausgestellt, dass es sehr einfach ist den Code zu verbessern, da sich der Fork fast wie ein eigenes Programm verhält. Es gibt auch einen kleinen Nachteil, wenn ein Fork abstürzt wird keine Fehlermeldung erzeugt. Somit musste ich während der Entwicklung bis der Prozess stabil gelaufen ist einige Debug Nachrichten im Code einfügen um zu sehen, ob der der Prozess noch läuft. Ansonsten war dies nur sichtbar dadurch, dass zum Beispiel ein Client sich ohne Grund vom Server getrennt hat.

2.4 Shared Memory

Bei einem Fork werden alle Variablen kopiert und sind dann unabhängig voneinander. Da ich aber einige Variablen, wie die Spieleranzahl oder das Spielfeld global und synchronisiert zwischen allen Forks haben wollte, musste ich einen Weg dafür finden. Die Lösung dafür ist Shared Memory [5]. Die sehr einfache und für meinen Anwendungsfall passende Lösung ist die Benutzung von einem anonymen Shared Memory. Dies konnte ich mit mmap realisieren.

2.5 Mutex oder Semaphore

Um die Variablen im Shared Memory vor gleichzeitigem Zugriff von den Forks zu schützen muss der Zugriff auf die Variable auf einen Prozess beschränkt werden. Dies kann mit Mutexen oder Semaphoren erreicht werden. Ich habe ein anschauliches und verständliches Beispiel gefunden [6], welches aufzeigt wie eine Variable im Shared Memory geschützt wird.

2.6 Client Strategie

Da ich weiss wie der Server funktioniert, war es für mich sehr einfach eine gute Client Strategie zu Entwickeln. Der Server sperrt zuerst alle Felder und prüft dann ob alle Felder vom selben Client besetzt sind. Da der Server beim Sperren von Feld x=0 y=0 anfängt und dann der Reihe nach durch geht, kann der Client dasselbe machen. Sobald der Client immer ein Feld einnimmt bevor der Server es sperrt hat er alle Felder bis alle gesperrt sind.

Die Funktion zuerst zu prüfen wem das Feld gehört habe ich vernachlässigt. Wenn das Feld bereits dem Client gehört, dann müsste er nur den Status abfragen. Wenn es ihm nicht gehören würde, wäre dies ein Geschwindigkeitsverlust, da er zwei Anfragen an den Server machen müsste. Zuerst um den Status zu erhalten und dann um das Feld effektiv einzunehmen.

3 Probleme, Lösung

3.1 Strings

Da es in C im Gegensatz zu C++ keine wirklichen Strings gibt, sondern nur Char-Arrays, war es eine Umgewöhnung, diese richtig zu Handhaben. Glücklicherweise gibt es aber die string.h library welche einige praktische Funktionen mit sich bringt [7].

Problem:

Das vergleichen von Zeichenketten ist nicht möglich.

```
if (buf == "HELLO\n")
```

Lösung:

Mit Verwendung der strcmp Funktion können zwei Zeichenketten miteinander verglichen werden.

```
if (strcmp(buf, "HELLO\n") == 0)
```

Problem:

Zeichenketten können nicht ohne weiteres zusammengefügt werden.

```
char size_string[64] = "SIZE " + number + "\n";
```

Lösung:

Mit Verwendung der strcat Funktion können zwei Zeichenketten miteinander verkettet werden.

```
char size_string[64] = "SIZE ";
strcat(size_string, number);
strcat(size_string, "\n");
```

3.2 Shared Memory

Damit auf eine Variable von verschiedenen Prozessen zugegriffen werden kann, muss sich diese an einem Ort befinden, der für alle zugänglich ist. Dafür bietet sich der Shared Memory Bereich an. Damit jedoch die Variablen korrekt dort gespeichert sind muss dies auch richtig gehandhabt werden [8].

Problem:

Das struct im Shared Memory funktioniert nicht.

```
playground = mmap(NULL, sizeof(*playground), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
playground = malloc(sizeof(field) * server.size);
```

Lösung:

Da es sich bei der Variable um einen struct handelt, welches ein dynamischer Char-Array enthält, muss der Bereich allokiert werden, bevor er in das Shared Memory gespeichert wird.

```
playground = malloc(sizeof(field) * server.size);
playground = mmap(NULL, sizeof(*playground), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

3.3 Locking

Auf Variablen im Shared Memory können verschiedene Prozesse gleichzeitig zugreifen, dies kann zu Inkonsistenzen und undefinierten Zuständen führen [9]. Deshalb müssen diese Variablen immer zuerst gesperrt werden, bevor auf diese Zugegriffen wird. Dadurch wird sichergestellt, dass nur ein Prozess gleichzeitig auf die Variable zugreift.

Problem:

Es wird eine Variable mit mehreren Prozessen bearbeitet.

```
if (buf == "HELLO\n")
```

Lösung:

Ein Lock erstellen bevor die Prozesse erstellt werden und bei jedem Zugriff auf die Variable diese Sperren.

```
sem_t *playcountlock = sem_open("playerSem", O_CREAT | O_EXCL, 0644, 0);
sem_unlink("playerSem");
sem_post(playcountlock);
// im fork()
sem_wait(playcountlock);
*playercount += 1;
sem_post(playcountlock);
```

3.4 Blocking

Die Clients beenden sich, wenn sie erfahren wer gewonnen hat. Die Prozesse des Servers beenden sich ebenfalls. Der Hauptprozess des Servers ist jedoch immer noch am Warten, bis sich neue Clients anmelden. Es gibt eine Möglichkeit diese wartende Funktion von blocking auf nicht-blocking umzustellen [10].

Problem:

Die accept funktion wartet, bis sich ein Client zum Server verbindet. Der Server weiss also erst bei eintreffen eines neuen Clients, dass das Spiel bereits fertig ist.

```
client_sock_fd = accept(sockfd, (struct sockaddr *)&client_addr,
&sin_size);
if (strcmp(end, "") != 0)
     break;
```

Lösung:

Die Funktion beim Erstellen des Sockets auf nicht-blocking umstellen. Nun kann abgefragt werden ob eine neue Verbindung vorhanden ist oder ob das Spiel beendet ist.

```
sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
client_sock_fd = accept(sockfd, (struct sockaddr *)&client_addr,
&sin_size);
if (client_sock_fd == -1)
{
      if (strcmp(end, "") != 0)
            break;
      continue;
}
```

3.5 Übertragung

Bei der Übertragung kann es vorkommen, dass nicht alle Zeichen abgeschickt werden um dieses Problem zu lösen wird abgefragt, wie viele Zeichen bereits gesendet wurden. Falls nicht alle Zeichen versendet wurden, werden diese nachgesendet.

Problem:

Es werden nicht alle Zeichen gesendet.

```
send(sock.fd, take, sizeof(take));
```

Lösung:

Mit der Funktion sendall wird sichergestellt, dass alle Zeichen gesendet werden.

```
sendall(sock.fd, take, sizeof(take));
int sendall(int s, char *buf, int len)
{
      int total = 0;
      int bytesleft = len;
      int n;

      while (total < len)
      {
            n = send(s, buf+total, bytesleft, 0);
            if (n == -1)
                  break;
            total += n;
            bytesleft -= n;
      }
      return n == -1 ? -1 : 0;
}
```

4 Fazit

Die Arbeit war sehr lehrreich und spannend. Da ich meist mit C++ arbeite, war es eine kleine Umgewöhnung nicht mehr all diese komfortablen Konstrukte zur Verfügung zu haben. Eine Debug Ausgabe etwa ist mit cout viel simpler, da ich mich nicht darum kümmern muss was die Variable für einen Typ hat. Auch habe ich die Strings etwas vermisst, welche einfacher zu vergleichen, zusammenfügen und manipulieren sind als die char Arrays.

Völlig neu war für mich die Erfahrung mit verschiedenen Prozessen zu arbeiten. Frühere Programme hatten immer einen regulären Ablauf. Ich habe gelernt wie ich mit Variablen umgehen muss, welche in verschiedenen Prozessen benötigt werden.

Ich habe zwar bereits etwas mit TCP in C++ gearbeitet, jedoch musste ich dort nur die Clientseite machen welche zum Server verbindet und einen String an diesen schicken. Bei dieser Aufgabe wiederum musste ich den Client und den Server Teil erstellen sowie eine reibungslose Kommunikation zwischen diesen. Es war dann wiederum interessant zu sehen wie schnell der Server mit all diesen Clients kommunizieren kann.

5 Anhang

5.1 Source Code

Der Source Code ist in folgendem GitHub Repository veröffentlicht:
https://github.com/samfisch3r/zhaw_concurrent_c_programming_fs_2015

5.2 Quellen

[1] «Beej's Guide to Network Programming,» [Online]. Available: http://www.beej.us/guide/bgnet/output/html/multipage/index.html.

[2] «Network Programming Sample,» [Online]. Available: https://github.com/thlorenz/beejs-guide-to-network-samples.

[3] «2 Dimensional Array of Struct,» [Online]. Available: http://stackoverflow.com/questions/3275381/how-to-implement-a-2-dimensional-array-of-struct-in-c.

[4] «Forking vs. Threading,» [Online]. Available: http://www.geekride.com/fork-forking-vs-threading-thread-linux-kernel/.

[5] «Shared Memory with Fork,» [Online]. Available: http://stackoverflow.com/questions/13274786/how-to-share-memory-between-process-fork.

[6] «Semaphore Example,» [Online]. Available: http://codepad.org/m1I9753u.

[7] «String Funktionen,» [Online]. Available: http://www.c-howto.de/tutorial-strings-zeichenketten-stringfunktionen.html.

[8] «Dynamic Array in shared struct,» [Online]. Available: http://stackoverflow.com/questions/14558443/c-shared-memory-dynamic-array-inside-shared-struct.

[9] «Lock Variable read/write,» [Online]. Available: http://stackoverflow.com/questions/18016806/in-which-cases-do-i-need-to-lock-a-variable-from-simultaneous-access.

[10] «Advanced Network Programming,» [Online]. Available: http://www.beej.us/guide/bgnet/output/html/multipage/advanced.html.