Seminararbeit

Limbo

Markus Müller
Vertiefungsseminar 2012/2013
LVA 703037-1

7. Februar 2013

Betreuer: Thomas Sternagel

Zusammenfassung (Englisch)

This seminar report concentrates on the programming language Limbo, developed in 1995/1996 in the Bell Labs and now maintained by Vita Nuova Holdings. It was designed for the distributed OS 'Inferno'[1], which was developed to achieve distributed systems on small computers. The syntax of Limbo is very similar to C and Ada and it is easy to read and understand. It features similar datatypes to other programming languages like C. Furthermore Limbo is absolutly portable because the compiler produces architecture independend object code which is interpreted by the *Dis* virtual machine (the virtual machine of Inferno) or just-in-time-compiled before runtime. In addition it supports concurrency with cheap processes and simple but comprehensive interprocess communication over the limbo type `channel`. Moreover it provides simple abstract data types and garbage collection. Nevertheless it does not support object orientation like Java or C#, therefore it is more hardware-based like C.

Inhaltsverzeichnis

1 Einleitung

Bisher wurden in der Welt der Informatik hunderte bis tausende Programmiersprachen entwickelt, welche mehr oder weniger alle ihre Daseinsberechtigung besitzen. Dabei lassen sich die Sprachen meist mindestens einer bestimmten Kategorie zuordnen. Objektorientierte, esoterische, Assembler, Datenbanksprachen oder imperative Programmiersprache, um nur einige zu nennen. Dabei erfüllen die meisten Sprachen eine Zweckmäßigkeit bzw. wurden aus einem ganz bestimmten Grund entwickelt, wie z.B. C für die Programmierung von Betriebssystemen und Anwendungen. Limbo wurde ebenfalls zweckmäßig entwickelt, nämlich für das Betriebssystem Inferno (näheres unter Abschnitt 2).[1] Als ein Nachfahre von C, gehört auch Limbo zu der Kategorie der imperativen Programmiersprachen. Imperativ deshalb, da der Programmcode aus einer Folge von Befehlen besteht, welche vom Prozessor in einer bestimmten Reihenfolge abgearbeitet werden sollen. Neben C gibt es aber noch weitere Merkmale welche sich in anderen Programmiersprachen wiederfinden wie etwa in Ada oder funktionalen Programmiersprachen wie OCaml. In den folgenden Abschnitten wird nun zuerst kurz auf die Entstehung der Programmiersprache Limbo eingegangen, danach wird die Syntax genauer analysiert mit Vergleichen zu bekannteren Programmiersprachen. Des weiteren wird näher auf die Vorzüge bzw. die unterstützten Technologien von Limbo, als auch auf die Existenzberechtigung der Sprache an sich eingegangen. Dieser Report basiert auf den folgenden Text-Quellen [2, 3, 4].

2 Entstehung

Die Programmiersprache 'Limbo' wurde 1995/1996 in den Bell Laboratories (auch bekannt unter Bell Labs) von Sean Dorward, Phil Winterbottom und Rob Pike entwickelt. In den Bell Labs wurden bereits zuvor und auch danach wichtige Errungenschaften erarbeitet, darunter auch das erste WaveLan (besser bekannt als WirelessLan oder WLan). Limbo wurde für das, von Dennis M. Ritchie entwickelte, verteilte Betriebssystem 'Inferno'[1] entworfen. In der Zwischenzeit ist nun die Firma Vita Nuova Holdings Ltd aus England für den Support und die Weiterentwicklung von Limbo sowie auch jene von Inferno zuständig.[2] Inferno kann direkt auf Hardware, als virtuelle Maschine auf einem bereits laufenden OS oder als einfache Applikation installiert werden. Der Compiler von Limbo erzeugt Objektcode, welcher auf jeder Maschine unabhängig von Architektur sowie virtueller Hardware ausgeführt werden kann, damit ist Limbo architektur- aber nicht plattformunabhängig. Das Einsatzgebiet von Limbo liegt vorwiegend in der Entwicklung von verteilten Systemen mit zusätzlichem Augenmerk auf kleinere Rechner. Merkmale von C (Rob Pike und Dennis M. Ritchie arbeiteten an der Entwicklung von C ebenfalls mit) sowie Ada, als auch von funktionalen Programmiersprachen sind zu entdecken, mehr dazu in Abschnitt 3.2.

[1] `http://www.vitanuova.com/inferno/`
[2] `http://www.vitanuova.com/company/index.html`

3 Verwendung

Zielsetzung der Entwicklung von Limbo war es, eine nebenläufige Programmiersprache zur Anfertigung von verteilten Systemen zu entwickeln, welche einfach zu lesen und zu verstehen ist. Limbo erfüllt diese Erwartung und soll zudem Aspekte von anderen hohen Programmiersprachen verwenden, welche aber bei Limbo, laut den Entwicklern, leichter zu erlernen sind. Dazu wird nachfolgend der Aufbau und die Syntax von Limbo analysiert.

3.1 Aufbau

Die nebenläufige Programmiersprache Limbo wird zur Entwicklung von verteilten Systemen basierend auf dem Inferno OS verwendet. Der vom Compiler produzierte Objektcode wird von der virtuellen Maschine *Dis* kurz vor der Ausführung interpretiert oder just-in-time in Maschinencode für die Zielarchitektur kompiliert. Dadurch erreicht Limbo eine komplette Architekturunabhängigkeit, welche der Entwicklung von verteilten Systemen zugutekommt.

Des weiteren ist Limbo bzw. eine Limbo-Applikation modular aufgebaut. Ein Limbo-Programm besteht aus einem oder mehreren Modulen: *.m - Dateien, welche eine Interfacedeklaration enthalten, welche von anderen Modulen verwendet werden können und *.b-Dateien welche die Implementierung beinhalten und auf die Deklarationen von *.m-Dateien zugreifen. Die Dateiendungen sind aber reine Konvention und könnten (sollten aber nicht) missachtet werden. Die Module werden dann zur Laufzeit dynamisch geladen. Greift ein weiteres Modul auf eine bereits geladene Modulinstanz zu, wird dieses nicht erneut in den Speicher geladen, sondern es wird das bereits geladene Modul aus dem Speicher verwendet. Dadurch wird verhindert dass Duplikate von Modulen im Speicher sind und dieser ist frei für andere Anfragen.

3.2 Syntax

Die Syntax von Limbo ähnelt besonders C und Ada (Erweiterung von Pascal). Besonders die Deklaration und Initialisierung von Variablen gleicht jener von Ada, während andere Konstrukte eher von C abstammen. Später wird anhand von zwei Beispielen(Listing 1 und Listing 2) die Syntax von Limbo genauer erklärt und auf Ähnlichkeiten zu anderen Sprachen eingegangen.

3.2.1 Datentypen

Limbo bietet unter anderem folgende Datentypen:

- `byte` (8-bit unsigned)
- `int` (32-bit signed)
- `big` (64-bit signed)
- `real` (64-bit floating point)

- `list`, ein vordefiniertes Konstrukt welches es ermöglicht Elemente zu verketten, dabei ähnelt die Liste stark den Listen aus funktionalen Programmiersprachen wie Ocaml oder Haskell

- `array`, vordefinierte Struktur zur dynamischen Verwaltung von mehreren Elementen gleichen Typs, vergleichbar mit den Arrays in C (z.B. `char[]`)

- `string`, Typ zur Verwaltung von Zeichenketten, in C ist so etwas nur mittels eines `char`-Arrays möglich

- `tuple`, geordnete Sammlung von mindestens zwei Elementen. Dabei können die Elemente verschiedene Typen haben. Funktionen können dadurch mehrere Werte verschiedenen Typs zurückgeben, kaum eine andere, nicht funktionale Programmiersprache unterstützt dies in solch einer Einfachheit. Tupel sind einfach umklammerte Ausdrücke bzw. Variablen, wobei die Anzahl der Elemente variabel sein kann aber mindestens zwei ist.

- `channel`, für Inter-Prozess Kommunikation, ähnlich zu den Pipes in C (genaueres unter Abschnitt 5)

- `adt` (abstract data type), kann Objekte verschiedenen Typs beinhalten und deklariert Funktionen welche damit arbeiten. Mehr dazu unter Abschnitt 4.4

- `module`, wird verwendet um ein Modul zu deklarieren um es so für andere Module verwendbar zu machen. Vergleichbar mit den Bibliotheken bzw. Headerfiles von C, konventionell in *.m-Dateien gespeichert

Genauere Informationen und weitere Datentypen sind der aktuellen Dokumentation von Limbo zu entnehmen.[3]

3.2.2 Deklaration und Zuweisung

Zuweisungen in Limbo erfolgen im Gegensatz zu vielen anderen Sprachen von rechts nach links wie etwa in Ada, d.h., dass zuerst die Variablen oder Objekte geschrieben werden, und dann der Typ der Variable(n). Des weiteren ist die Zuweisung intelligenter als jene von C. Das folgende kurze Beispiel soll dies verdeutlichen.

`p: Point;` Die Variable `p` vom Typ `Point` wird deklariert. Der Typ `Point` ist ein Objekt vom Typ `adt` und besitzt zwei Werte welche die Koordinaten eines Punktes beinhalten.

`x, y: int;` Die Variablen `x` und `y` bekommen Typen `int`. Diese Deklaration entspricht `int x, y;` in C.

`(x, y) = p;` Die zwei Werte in `p` werden automatisch auf `x` und `y` aufgeteilt, dafür muss nur die Anzahl der Elemente der linken Seite mit jener des `adt` oder Tupels der rechten Seite übereinstimmen.

[3] `http://doc.cat-v.org/inferno/4th_edition/limbo_language/limbo`

Im Allgemeinen erfolgt die Zuweisung von Werten oder Objekten analog zu den meisten anderen Sprachen wie C oder auch Java. Links vom = steht die Variable welche den Wert bekommen soll, und rechts ein Wert oder Objekt welches deklariert und initialisiert wurde. Aber auch komplexere Zuweisungen sind möglich wie `x: int = 1;`, welche `x` als Integer deklariert und sofort den Wert `1` zuweist. Dabei geht Limbo noch einen Schritt weiter und erlaubt auch Konstrukte wie `x := 1`, das den gleichen Zweck erfüllt wie das Beispiel vorher, da Limbo automatisch den Typ des Wertes ableitet und die Variable richtig deklarieren kann.

3.2.3 Keywords

Wie auch in anderen Programmiersprachen gibt es in Limbo reservierte Keywords, welche weder als Variablenname noch als Bezeichner für Funktionen verwendet werden dürfen, da sie bereits eine Bedeutung haben bzw. Funktion erfüllen. Eine Gruppe der nicht verwendbaren Keywords sind alle Datentypen welche Limbo unterstützt. Im folgenden werden einige weitere Keywords, welche es in anderen Sprachen nicht oder nur in anderer Form gibt, aufgelistet und kurz erklärt.

- `alt`: Wählt einen von mehreren Befehlsblöcken nach der Lesbarkeit von Kanälen aus und führt diesen aus. Vergleichbar mit if-else oder switch-case in C oder Java, nur hängt die Bedingung mit einem Kanal zusammen.
- `con`: Ermöglicht es Konstanten Namen zu geben, z.B. `Seven: con 7;`, Konstanten können nicht durch Zuweisungen verändert werden.
- `cyclic`: Hiermit können zyklische Datenstrukturen erstellt werden. Diese werden vom Garbage-Collector entfernt, wenn sie nur mehr auf sich selbst referenzieren.
- `fn`: Dient zur Deklaration von Funktionen.
- `hd`: Operator, gibt das erste Element (`hd` = head) zurück. Listen sind dabei ähnlich strukturiert wie in funktionalen Programmiersprachen wie etwa OCaml. Der Aufruf `hd (a :: l)` gibt `a` zurück, wobei `a` ein Element ist und `l` der Rest der Liste.
- `len`: Gibt die Länge einer Liste oder eines Arrays zurück.
- `nil`: Gegenstück zu `NULL` in C.
- `self`: Ähnlich zu der `this`-Referenz in Java. Wird in `adt`-Objekten verwendet um z.B. bei Funktionsaufrufen auf das aufrufende Objekt zu referenzieren
- `tl`: Gegenstück zu `hd`, gibt die Liste ohne das erste Element zurück (`tl` = tail)

3.2.4 Hello World

Im folgenden wird Zeile für Zeile der Code näher erläutert. Zeile 1 gibt an, dass in diesem Modul das Programm HelloWorld implementiert wird, um so anderen Modulen sofort

```
1  implement HelloWorld;
2
3  include "sys.m";
4    sys: Sys;
5
6  include "draw.m";
7
8  HelloWorld: module
9  {
10   init: fn(nil: ref Draw->Context, nil: list of string);
11 };
12
13 init(nil: ref Draw->Context, nil: list of string)
14 {
15   sys = load Sys Sys->PATH;
16   sys->print("Hello World!\n");
17 }
```

Listing 1: Ein Hello World Programm in Limbo

ersichtlich zu machen, um welches Modul es sich handelt. Zeile 3 sowie 6 inkludieren jeweils ein Modul (`sys.m` um die Funktoin `print()` zu verwenden, `draw.m` würde benötigt werden, wenn Grafiken ausgegeben werden), gleichzusetzen dem `#include`-Befehl in C. Zeile 4 deklariert die Variable `sys` vom Typ `Sys` aus dem Modul `sys.m`, `sys` hat hier noch keinen konkreten Wert sondern `nil`. Von Zeile 8 bis 11 erfolgt die Interfacedeklaration des Moduls. Zuerst wird dem Programm HelloWorld der Typ `module` zugewiesen, damit evtl. andere Programme darauf zugreifen können. Danach werden die Funktionen deklariert, was ähnlich zur Variablendeklaration geschieht. Zuerst kommt der Name der Funktion und nach dem Doppelpunkt der Typ, also `fn` für „function“. In diesem Beispiel wird nur die `init`-Funktion (gleich zu stellen mit der `main()`-Funktion in C) deklariert, inklusive den Parametern, welche übergeben werden können. Dabei ist zu beachten, dass für die Variablen das Keyword `nil` eingesetzt wurde. Übergebene Argumente an diese Funktion werden so unverwendbar gemacht, da das Keyword `nil` keine Werte annehmen kann. Grund dafür ist, dass bei Limbo zwei Argumente für die `init`-Funktion Konvention sind, ungeachtet dessen, ob sie benötigt werden. Die Typen aber sollten trotzdem korrekt angegeben werden, in diesem Fall einmal eine Referenz (also ein Pointer) auf ein Element von `Draw`, `ref` steht hier für den Zeiger, wie `*` in C, und der Pfeil `->` für den Zeiger auf das Element des `adt Draw`. Das zweite Argument ist eine Liste vom Typ `string`.

In den Zeilen 13 bis 17 erfolgt die Implementierung. Dabei wird zuerst `sys` ein Wert zugewiesen, welcher vom Sys-Modul geladen wird. Danach kann auf die `print()`-Funktion zugegriffen werden und ein `"Hello World!"` wird ausgegeben.

```
1  implement BeerBottles;
2
3  include "sys.m";
4    sys: Sys;
5  include "draw.m";
6    draw: Draw;
7
8  BeerBottles: module
9  {
10   init: fn(ctxt: ref Draw->Context, argv: list of string);
11 };
12
13 init(ctxt: ref Draw->Context, argv: list of string)
14 {
15   sys = load Sys Sys->PATH;
16   for (int b = 99; b > 0; b--) {
17     sys->print("%d bottle(s) of beer on the wall,\n", b);
18     sys->print("%d bottle(s) of beer.\n", b);
19     sys->print("Take one down, pass it around,\n");
20     sys->print("%d bottle(s) of beer on the wall.\n\n", b-1);
21   }
22 }
```

Listing 2: Dieses Programm gibt den Song „99 bottles of beer“ aus

3.2.5 99 Bottles of Beer

Die ersten Zeilen dieses Programms sind analog zu Hello World. Kleinere Unterschiede wie konkrete Variablen anstelle des Keywords `nil` im Funktionskopf sind zu erkennen. Der erste markante Unterschied begegnet uns in Zeile 16, wo die `for`-Schleife beginnt. Man kann sofort erkennen, dass sie sich kaum von anderen Schleifen aus anderen Sprachen wie C oder Java unterscheidet. Der Schleifenkopf beginnt mit der Deklaration und Initialisierung der Iterationsvariable, gefolgt von der Abbruchbedingung und zuletzt der In- oder Dekrementierung der Variable. Des weiteren ist erkennbar, dass sich die `print()`-Funktion ähnlich wie `printf` von C verhält. Man kann im String mit `%` Platzhalter definieren und dahinter als weitere Argumente Variablen oder Werte für diese Platzhalter angeben.

Die Trennung von Befehlen in Limbo erfolgt wie in den Beispielen zu erkennen durch ein einfaches Semikolon „`;`“.

4 Features

Nicht nur die Syntax von Limbo ähnelt der von C, Ada und Java, auch die meisten Features sind vergleichbar zu jenen, die man in anderen Sprachen findet. Ein herausragendes Feature von Limbo ist die einfache Nebenläufigkeit, welche darum in Abschnitt 5 eigens behandelt wird. Im folgenden werden die anderen Features kurz zusammengefasst.

4.1 Modulare Programmierung

Wie bereits erwähnt baut Limbo auf die modulare Programmierung auf. Module werden zur Laufzeit in den Speicher geladen, wobei immer nur eine Instanz pro Modul im Speicher vorhanden sein kann. Der modulare Aufbau fördert die Wartbarkeit von Programmen sowie die Erweiterbarkeit. Ähnlich zu den Modulen kann man die Bibliotheken bzw. selbst definierte Headerfiles in C sehen, wobei Module meist auch eine Implementierung besitzen.

4.2 Type Checking

Type Checking erfolgt bei Limbo zur Compile- als auch Laufzeit, was es prinzipiell ermöglicht generische Typen zu verwenden, welche erst zur Laufzeit bekannt werden. Andere Sprachen wie C setzen nur auf statisches Type Checking (also zur Compilezeit). Java hingegen unterstützt sowohl statisches als auch dynamisches Type Checking.

4.3 Garbage Collection

Limbo besitzt einen eigenen Garbage Collector, welcher automatisch Speicher frei gibt der nicht mehr referenziert wird, ähnlich wie jener in Java. Somit muss man sich nicht Gedanken darüber machen, allozierten Speicher nach dem Gebrauch wieder freizugeben, wie es etwa in C z.B. mit der Funktion `free()` getan werden muss/sollte.

4.4 Abstract Data Types (ADT)

In Limbo können simple Abstrakte Datentypen verwendet werden, welche mehrere Objekte besitzen können und eigenständige Funktionalität aufweisen. Ähnlich zu den Klassen in Java, wird so Funktionalität in eine Datenstruktur gegeben und über ein Interface zugegriffen, welches keine Kenntnisse der eigentlichen Implementierung verlangt. Dies stimmt mit dem Grundsatz der modularen Programmierung von Limbo überein, da so Änderungen an der Implementierung des Datentyps keine Anpassungen in jenem Programm verlangen, welche ein ADT-Objekt benutzen oder ein anderes Modul referenzieren.

5 Nebenläufigkeit

Da Limbo zur Entwicklung von verteilten Systemen verwendet wird, ist Nebenläufigkeit ein zentraler Aspekt. Limbo unterstützt deshalb Multi-Threading sowie einen einfachen

Synchronisationsmechanismus über Variablen des Typs `channel`. Dabei ist die Erzeugung von Threads sowie die dazugehörige Synchronisation um ein vielfaches einfacher gestrickt als es in C der Fall ist.

5.1 Threads

Das Erstellen von Threads ist in Limbo sehr einfach und die Prozesse an sich sind sehr sparsam im Umgang mit den zur Verfügung stehenden Ressourcen. Um einen Thread zu starten, benötigt man die Funktion `spawn` gefolgt von der Funktion welche der Thread im Laufe seines Zyklus ausführen soll. Die Funktion `spawn` zum Erzeugen von Threads ist in jedem Modul ohne vorheriges einbinden eines anderen Moduls verfügbar. Beispielaufruf: `spawn run(arg);`, ein neuer Thread wird erzeugt und führt die Methode `run` mit übergebenem Argument `arg` aus.

5.2 Channel

Zur Synchronisation sowie Kommunikation zwischen Prozessen verwendet Limbo Instanzen vom Typ `channel`. Man kann einen Kanal mit den Pipes in C vergleichen. Bei der Initialisierung eines Kanals kann direkt angegeben werden, welcher Datentyp später über diesen Kanal geschickt wird. Dieser Kanal kann darauf als Argument beim erzeugen eines Threads übergeben werden. Übergibt man diesen mehreren Threads können alle über diesen einen Kanal kommunizieren bzw. synchronisiert werden. Ein Thread kann auch mehr als nur einen Kanal zur Kommunikation/Synchronisation verwenden.

5.3 Beispiel

Listing 3 zeigt anhand eines trivialen Beispiels, wie zwischen zwei Threads mittels dem Kanal `sync` synchronisiert werden kann. Der Beginn des gesamten Moduls scheint hier nicht auf, da dieser bis zur init-Funktion analog zu Listing 1 und Listing 2 ist. Das Programm an sich bekommt Werte als Kommandozeilenparameter und gibt diese hintereinander mit einer Pause von einer Sekunde aus. Dabei ist ein zweiter Prozess dafür zuständig dem Hauptprozess Bescheid zu geben wann eine Sekunde vorbei ist.

In Zeile 7 wird der Integer-Kanal `sync` deklariert, welcher dann zusammen mit der Anzahl an Werten in Zeile 9 der Funktion `timer` übergeben wird, welche vom neu erzeugten Thread (mittels `spawn`) ausgeführt werden soll. Wenn Konstrukte wie etwa Kanäle oder Listen bestimmte Typen haben können, kommt in der Deklaration nach der Art von Konstrukt (`chan`) ein `of` gefolgt vom gewünschten Typ, deshalb `chan of int`. In den Zeilen 12 bis 17 erfolgt die Ausgabe, wobei Zeile 13 für die Synchronisation zuständig ist. Hier wartet der Hauptprozess, welcher nun unabhängig vom `timer`-Prozess weiterläuft, bis ein Wert in den Kanal geschrieben wird. Dafür sorgt der Pfeil `<-`, welcher vom Kanal weg zeigt. Ein Pfeil dieser Art, egal ob links oder rechts, zeigt auf, dass auf etwas im Ursprung zugegriffen wird. Ein Pfeil nach links heißt es soll etwas entnommen und gegebenenfalls dem Ziel zugewiesen werden, da es sich um einen Kommunikationsoperator handelt, ein Pfeil nach rechts bedeutet, es wird auf ein Element

```
1  .
2  .
3  init(nil: ref Draw->Context, argv: list of string)
4  {
5    sys = load Sys Sys->PATH;
6
7    sync := chan of int;
8    n := len argv;
9    spawn timer(sync, n);
10
11   sys->print("Command Line Parameters\n");
12   for (i := 0; i < n; i++) {
13     <-sync;
14     sys->print("%d: %s\n", i, hd argv);
15     argv = tl argv;
16   }
17 }
18
19 timer(sync: chan of int, n: int)
20 {
21   for (i := 0; i < n; i++) {
22     sys->sleep(1000);
23     sync <-= 1;
24   }
25 }
```

Listing 3: Synchronisation über Kanal sync zwischen Haupt- und Nebenprozess

oder eine Funktion zugegriffen. Sollte nun `sync` leer sein so wartet der Prozess bis etwas in den Kanal gelegt wird. Der Wert könnte wie bereits erwähnt auch einer Variable mit `variablenname:= <-sync;` zugewiesen werden, jedoch wird er nicht mehr weiter benötigt. Danach wird das erste Element der Parameter (`hd argv`) ausgegeben und aus der Liste entfernt (`argv = tl argv`). Der andere Thread macht nichts anderes als `n`-mal eine Sekunde zu warten (`sys->sleep(1000);`) und darauf eine 1 in den Kanal zu schreiben um so dem Main-Thread zu signalisieren, dass die Sekunde vergangen ist. Sobald dies `n`-mal wiederholt wurde beendet sich der Thread.

Im Gegensatz zur recht simplen Nebenläufigkeit von Limbo steht jene von C, welche an einem einfachen Beispiel in Listing 4 gezeigt wird. Die in C einfachste Art Prozesse zu erzeugen ist die Funktion `fork()` (Zeile 11), welche einfach einen neuen Prozess erzeugt, welcher eine Kopie des ursprünglichen Prozesses ist und dessen Prozessnummer zurückgibt. Der neue Kindprozess startet direkt nach dem `fork` Befehl, also in Zeile 12. Um nun dem Kindprozess eigene Aufgaben zuweisen zu können, muss ständig die Prozessnummer überprüft werden (Zeile 17), da beide Prozesse den gleichen Code ausführen.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  main(){
6    int     fd[2];
7    pid_t   childpid;
8
9    pipe(fd);
10
11   childpid = fork();
12   if(childpid == -1){
13    perror("fork");
14    exit(1);
15   }
16
17   if(childpid == 0){
18    /* Child process closes up input side of pipe */
19    close(fd[0]);
20   }else{
21    /* Parent process closes up output side of pipe */
22    close(fd[1]);
23   }
24   .
25   .
26 }
```

Listing 4: Pipes und Threads in C

Um zwischen den Prozessen zu kommunizieren wird zusätzlich eine Pipe benötigt, welche vor einem `fork()` mit `pipe()` erzeugt wird (Zeile 9). Will dann z.B. der Elternprozess Daten vom Kind bekommen, müssen die entsprechenden Enden der Pipes geschlossen werden (Zeilen 19 und 22). Nun erst ist es möglich, Byte für Byte Daten vom Kindprozess an den Elternprozess zu schicken. Mit den Kanälen in Limbo ist es sogar möglich mehrere verschiedene Typen zu verschicken (z.B. ist ein `chan of (int, string);` ein Kanal welcher es ermöglicht, sowohl Integer als auch ganze Strings als `tuple` zu versenden).

Das Beispiel aus Listing 2 kann ebenfalls nebenläufig programmiert werden. Eine mögliche Implementierung finden Sie auf der Website `99-bottles-of-beer.net`.[4]

[4] `http://99-bottles-of-beer.net/language-limbo-1892.html`

6 Ausblick und Fazit

Das verteilte Betriebsystem Inferno benötigte aufgrund seiner schlanken Struktur und der Möglichkeit, das OS direkt auf Hardware, als virtuelle Maschine oder als einfache Applikation zu installieren, eine architekturunabhängige Programmiersprache. Dies erfüllt Limbo da der Compiler zuerst nur Objektcode erzeugt und erst kurz vor der Ausführung von der virtuellen Maschine *Dis* interpretiert bzw. in Maschinencode kompiliert wird. Zudem sollten Programme auf Inferno möglichst sparsam mit Ressourcen umgehen, da verteilte Systeme mit Inferno auch auf kleineren bzw. schwächeren Rechnern performant und stabil laufen sollten. Auch dies erfüllt Limbo ohne Zweifel, da die Prozesse wenig Ressourcen verbrauchen und auch der Speicher durch den Garbage Collector (siehe Abschnitt 4.3) möglichst effizient genutzt wird. In Sachen objektorientiertem Programmieren hinkt Limbo nur mit der Möglichkeit von simplen ADTs (Abschnitt 4.4) anderen Sprachen hinterher. Java aber wäre z.B. für den Gebrauch auf Inferno viel zu verschwenderisch und bei einer hardwarenahen Sprache wie C müsste man auf einige Features verzichten welche Limbo einfach und verständlich zur Verfügung stellt. Jedoch wäre C die bessere Option als eine höhere Sprache wie Java oder C++, da diese unter dem schlanken OS entweder gar nicht oder sehr unperformant funktionieren würde. Somit hat Limbo seine Daseinsberechtigung, auch wenn Inferno an sich eher selten zum Einsatz kommt, und somit auch Limbo nur begrenzt Verwendung findet. Eines von sehr wenigen Projekten ist *gitfs*.[5] Dieses Projekt ist ein Interface um eine Verwendung von Git-Repositories unter Inferno zu ermöglichen und diese einfach und komfortabel zu verwalten. Ansonsten sind kaum Projekte dieser Art aufgrund der Unpopularität von Inferno und Limbo zu finden.

[5] `https://github.com/manzur/gitfs`

Literatur

[1] Inferno (operating system).
http://en.wikipedia.org/wiki/Inferno_(operating_system).
Dezember 2012.

[2] Limbo (programming language).
http://en.wikipedia.org/wiki/Limbo_(programming_language).
Dezember 2012.

[3] Vita nuova - limbo.
http://www.vitanuova.com/inferno/limbo.html.
Dezember 2012.

[4] D. M. Ritchie. The limbo programming language.
http://www.vitanuova.com/inferno/papers/limbo.html.
Dezember 2012.