Belegarbeit

Eine Shell für L4Env

Aaron Pohle

26. Mai 2008

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter: Björn Döbel, Michael Roitzsch

Zusammenfassung

Betriebssysteme bestehen heutzutage aus vielen einzelnen Programmen, welche verschiedene Dienste anbieten. Um solch ein heterogenes System verwalten und bedienen zu können, wird eine flexible Schnittstelle zum System benötigt. Eine Shell – und die damit verbundene Skriptsprache – stellt solch eine Schnittstelle zur Verfügung.

Ziel des Belegs war es, eine skriptbare Shell für das L4 Environment (*L4Env*) zu schaffen. Diese soll es ermöglichen Bibliotheksfunktionen aufzurufen, um damit Anwendungen unter L4 debuggen bzw. L4 Systeme administrieren zu können. Ausgehend von Anforderungen an solch eine Shell/Skriptsprache, wählte ich Python um eine L4 Shell zu bauen. Das Steuern von L4 Programmen kann mittels Bibliotheksfunktionsaufrufen oder IPC erfolgen. Meine Belegarbeit zeigt wie beide Möglichkeiten in Python unter L4 genutzt werden können.

Belegaufgabe: Administration and debugging shell for *L4Env*

Zentrales Ziel der Aufgabe ist es, eine skriptbare Shell für das L4 Environment (*L4Env*) zu schaffen. Diese Shell soll dazu dienen, Anwendungen im *L4Env* zu debuggen bzw. laufende L4 Systeme zu administrieren. Dazu ist es notwendig, aus der Shell heraus Aufrufe an L4 Server auszuführen. Hierfür soll im Rahmen einer Skriptsprache die Möglichkeit geschaffen werden, Server-Aufrufe zu tätigen.

Speziell ist es dabei erforderlich, Nutzer-Eingaben auf im System vorhandene Services und deren Interfaces abzubilden, die notwendigen Client-Bibliotheken dynamisch nachzuladen, die vom Nutzer eingegebenen Argumente an den Client-Funktionsaufruf weiterzugeben und die Rückgabewerte des Aufrufs in geeigneter Form auszugeben.

Die Aufgabe umfasst insbesondere folgende Schritte:

- Auswahl einer geeigneten Skriptsprache (anhand Portierbarkeit und Eignung für L4 bzw. Anbindbarkeit an das *L4Env*)
- Portierung der Skriptsprache und einer Eingabe-Shell auf das *L4Env* (ggf. unter Verwendung vorhandener Terminals des *L4VFS*)
- Realisierung der Administrations-Shell in der Skriptsprache:
  - Abbildung der Nutzereingaben auf im System vorhandene Services
  - Marshalling komplexer Funktionsparameter (z.B. L4 Thread IDs, Datenstrukturen)
  - Dynamisches Nachladen der erforderlichen Client-Bibliotheken und Ausführen der notwendigen C-Funktionen.
  - Geeignete Darstellung der erhaltenen Rückgabewerte

Einführung

Heutige Betriebssysteme bestehen aus vielen Teilen, meist einzelnen Programmen, die unterschiedliche Funktionalitäten anbieten. Insgesamt ergibt sich ein heterogenes System, bestehend aus verschiedenen Programmen, die jeweils eine eigene Schnittstelle zur Nutzung ihrer Funktionen bereitstellen.

Zur Bedienung solcher Systeme eignen sich Shells/Skriptsprachen. Mit Hilfe dieser können Abläufe automatisiert, Administrationsaufgaben durchgeführt und Schnittstellen getestet werden. Insgesamt stellen Shells/Skriptsprachen eine Schnittstelle für den Benutzer zum System zur Verfügung.

Denkbare Beispiele zur Nutzung von Shells sind das Starten von Programmen, das Arbeiten mit Dateien und die Möglichkeit, Ein- und Ausgaben von Programmen umzulenken. Mit Hilfe von Skriptsprachen können Abläufe automatisiert werden. In verteilten Systemen können zum Beispiel Namensdienste abgefragt und mit dieser Information andere Dienste adressiert und dadurch in Anspruch genommen werden.

In ersten Teil “Grundlagen” (Kapitel 1) meiner Arbeit beschäftige ich mich damit, eine passende Shell/Skriptsprache zu finden. Dazu stelle ich ausgewählte Skriptsprachen kurz vor und beschreibe die Zielumgebung der Portierung. Im zweiten Teil “L4Python” (Kapitel 2) diskutiere ich zuerst Anforderungen an eine Shell für L4. Danach stelle ich die Portierung von Python auf L4 vor. Um effektiv mit der Shell unter L4 arbeiten zu können, muss es möglich sein, mit anderen Programmen zu kommunizieren. Dies kann mit Bibliotheksfunktionsaufrufen realisiert werden. Im Kapitel 3 stelle ich Möglichkeiten vor, Python mit Bibliotheken zu erweitern. Dazu beschreibe ich zuerst allgemein Ansätze zur Erweiterung von Python mit L4 Bibliotheken sowie danach deren Umsetzung an verschiedenen Beispielen.

Die Qualität und Geschwindigkeit der Portierung bewerte ich in Kapitel 4 “Auswertung”. Abschließend gebe ich im Kapitel 5 einen kurzen Ausblick und fasse die Ergebnisse meiner Arbeit im Kapitel 6 “Zusammenfassung” zusammen.

Inhaltsverzeichnis

Kapitel 1

Grundlagen

In diesem Kapitel lege ich Grundlagen dar, auf denen nachfolgende Kapitel aufbauen. Zuerst beschreibe ich allgemein wichtige Eigenschaften von Skriptsprachen. Danach stelle ich mögliche Skriptsprachen vor, wobei für diese Arbeit relevante Funktionalitäten vorgestellt werden. Abschließend stelle ich die Zielumgebung der Portierung – das *L4Env* – und die POSIX-Schnittstelle vor.

1.1 Skriptsprachen

In einem Artikel aus dem Jahr 1998 beschreibt John K. Ousterhout Skriptsprachen:

> Scripting languages are designed for different tasks than system programming languages, and this leads to fundamental differences in the languages. System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements such as words of memory. In contrast, scripting languages are designed for gluing: they assume the existence of a set of powerful components and are intended primarily for connecting components together. System programming languages are strongly typed to help manage complexity, while scripting languages are typeless to simplify connections between components and provide rapid application development. [Ouster98]

John. K. Ousterhout unterscheidet Skriptsprachen und systemnahe Programmiersprachen. systemnahe Programmiersprachen dienen zum Implementieren von Datenstrukturen und Algorithmen mit Hilfe von primitiven Datentypen Skriptsprachen verfügen dagegen über mächtige Komponenten und ermöglichen einen schnellen Entwicklungsprozess.

Im Unterschied zu systemnahen Programmiersprachen wird bei Skriptsprachen meist Quellcode oder Bytecode (erzeugt durch Kompilierung aus Quellcode) interpretiert. Der Interpreter liegt o.B.d.A. meist in nativem Maschinencode vor und wird direkt ausgeführt.

Merkmale von Skriptsprachen sind dynamische Typisierung (der Typ einer Variablen wird vom Laufzeitsystem selbst erkannt), automatische Speicherverwaltung (ein Garbage Collector gibt Speicher, der nicht mehr benötigt wird, frei) und Objektorientierung.

Shells sind interaktive Interpreter einer Skriptsprache. Diese führen Befehle der Skriptsprache direkt aus, nachdem der Benutzer diese eingegeben hat. Shells eignen sich für Beobachtung, Steuerung und Entwicklung von Systemen.

Eine mögliche Unterteilung der Skriptsprachen und Interpreter ist:

- Kommandozeileninterpreter mit Skriptsprache; dienen zum Steuern von Betriebssystemen (Linux, Unix, ...), z.B. Bash
- Skriptsprachen zum Verbinden mehrerer Programme mittels Ein- und Ausgabe, z.B. awk, sed

- höhere Skriptsprachen mit Objektorientierung, I/O-Funktionalität (sind meist gleichwertig zu hohen Programmiersprachen wie Java, C++, C), z.B. Perl, Python.

Es gibt auch Skriptsprachen ohne Shell bzw. Shells ohne Skriptsprache.

Wichtig ist die Möglichkeit, Skriptsprachen erweitern zu können. Eine Erweiterung bedeutet, dass Quellcode, geschrieben in einer anderen Programmiersprache, eingebunden werden kann. Ein denkbares Beispiel für die Erweiterung einer Skriptsprache ist das Hinzufügen von Bibliotheken. Diese Bibliotheken können zum Beispiel das Erstellen grafischer Anwendungen ermöglichen. Durch Einbinden der Bibliothek in eine Skriptsprache, können die Bibliotheksfunktionen genutzt werden.

Im Gegensatz zur Erweiterung steht die Einbettung einer Skriptsprache. Bei der Einbettung wird die Skriptsprache in eine Programmiersprache eingebunden. Dadurch können Skripte (Quellcode, geschrieben in einer Skriptsprache) aus der Programmiersprache heraus direkt aufgerufen werden. Benutzt wird die Einbettung beispielsweise, um Befehlsshells in Anwendungen zu integrieren und so die Anwendung steuern und beobachten zu können.

Im Folgenden stelle ich kurz ein paar Skriptsprachen vor und erläutere wichtige Funktionalitäten.

1.1.1 Bash

Die Bash (Bourne-again-shell)[Bash08] ist eine Unixshell, die als Teil des GNU (GNU is not UNIX) Projektes entwickelt wird. Geschrieben wurde die Bash in den 80er Jahren im Wesentlichen von Brian Fox und Chet Ramey und ist kompatibel zur originalen Bourne-Shell.

Die Bash ist ein interaktiver Kommandozeileninterpreter, der die Schnittstelle zu Unix/Linux-Systemen bildet. Im Wesentlichen dient die Bash zum Starten von Programmen und dem Umlenken von Ein-/Ausgaben. Weiterhin verfügt die Bash über eine eingebaute Skriptsprache, die sich zum Starten von Programmen eignet. Die Bash ist unter der GPL lizenziert und frei verfügbar.

1.1.2 Interactive Ruby Shell

Ruby[Ruby08] ist ist eine interpretierte, objektorientierte Programmiersprache, die von Yukihiro Matsumoto 1993 in Japan entwickelt wurde. Die Interactive Ruby Shell ist ein Kommandozeileninterpreter für Ruby.

Eine Erweiterung von Ruby ist durch Module möglich, die dynamisch zur Laufzeit geladen werden. Weiterhin kann Ruby auch in andere Programme eingebettet werden. Ruby ist freie Software und steht unter der GPL.

1.1.3 Lua

Lua[Lua08] ist eine prozedurale, objektorientierte Skriptsprache, die 1993 von der Computer Graphics Technology Group der Pontifikalen Katholischen Universität von Rio de Janeiro in Brasilien entwickelt wurde.

Lua ist zum Einbetten in Programme konzipiert und als Bibliothek in ANSI-C implementiert. Es ist möglich, C-Code und Lua-Code zu mischen; die Variablen und Funktionen sind in beide Richtungen erreichbar. Lua verfügt über einen eigenständigen Interpreter. Dieser benutzt die Lua-Bibliothek, um Lua-Code auszuführen.

Weiterhin ist es möglich, Lua mit Bibliotheken, welche in anderen Sprachen geschrieben sind, zu erweitern. Dazu müssen lediglich Wrapper geschrieben werden, die zur Lua-Bibliothek hinzugefügt werden. Lua ist freie Software und wurde bis zur Version 4 unter einer eigenen BSD-Lizenz, ab Version 5 unter der MIT-Lizenz veröffentlicht.

1.1.4 Perl

Larry Wall entwarf 1987 die prozedurale, objektorientierte und interpretierte Programmiersprache Perl[Perl08]. Diese wurde ursprünglich als Werkzeug zur Verarbeitung und Manipulation von

Textdateien entwickelt und bringt viele Elemente von sed, awk, sh, C und vielen anderen Sprachen mit.

Perl kann durch nachladbare Module erweitert werden. Diese werden in der Sprache XS (eXternal Subroutines) geschrieben und beinhalten Wrapper zum Aufruf von Bibliotheksfunktionen. Umgekehrt kann Perl auch in Programme eingebettet werden. Perl ist freie Software und kann unter der GPL (GNU General Public License) oder Artistic License verbreitet werden.

1.1.5 Python

Python[Python08] ist eine interpretierte, objektorientierte und prozedurale Programmiersprache. Anfang der 90er Jahre wurde Python von Guido van Rossum am Centrum voor Wiskunde en Informatica in Amsterdam als Nachfolger für die Programmier-Lehrsprache ABC entwickelt. Es war ursprünglich für das verteilte Betriebssystem Amoeba gedacht.

Durch das Python API (Application Programmers Interface) ist das Einbetten oder Erweitern von Python sehr einfach möglich. Durch Einbetten von Python in Programme können diese mit in Python geschriebenen Skripten erweitert werden.

Mit Hilfe von Erweiterungsmodulen kann Python erweitert werden. Diese Module können Funktionalität zu Python hinzufügen oder vorhandene Konstrukte bzw. Standardfunktionen ersetzen. Python ist unter der *Python license* lizenziert, die freie Benutzung und Verbreitung für private und kommerzielle Projekte erlaubt.

1.2 *L4Env*

Das *L4Env*[L408] ist Teil des Dresden Real-Time Operating System und bildet eine Schnittstelle zu L4 kompatiblen Mikrokernen. Die Idee des *L4Env* ist, eine Umgebung zur Entwicklung von L4 Programmen bereitzustellen. Das *L4Env* basiert auf Servern, die Dienste bereitstellen und Bibliotheken, mit denen diese Dienste in Anspruch genommen werden können (siehe Abbildung 1.1).

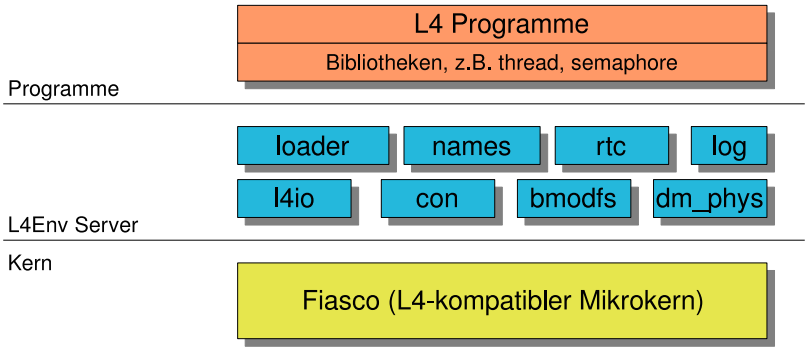

Abbildung 1.1: Das *L4Env*

Die Server kommunizieren über den IPC-Mechanismus (interprocess communication) von L4. Da die manuelle Erzeugung des Codes für die Kommunikation sehr aufwändig und fehlerträchtig ist, wird zur Erzeugung des Stubcodes ein IDL-Compiler benutzt. Notwendig ist lediglich eine Beschreibung der Schnittstelle mit Hilfe von IDL (interface description language). Der IDL-Compiler generiert daraus Server- und Client-Stubcode. Mit Hilfe des Clientstubcodes können Bibliotheken implementiert werden, die Dienste verschiedener Server anbieten. Durch Integration dieser Bibliotheken in Skriptsprachen können von Servern bereitgestellte Dienst in Skripten benutzt werden.

Das *L4Env* bietet Client-Bibliotheken und Server für unterschiedliche Dienste. Einige in der Portierung verwendete Bibliotheken und Server möchte ich kurz vorstellen. Als C-Bibliothek dient

die uClibc[uClibc08], eine kleine C-Bibliothek für eingebettete Systeme. Zur Nutzung von Threads stellt die Threadbibliothek (`thread`) Funktionen bereit, mit denen native L4 Threads erzeugt werden können. Weiterhin steht eine Bibliothek für Semaphore (`semaphore`) zur Verfügung.

Ein L4 System ähnelt hinsichtlich der Kommunikation einem verteilen System, in solchen benötigen die Komponenten Adressierungmöglichkeiten. Das *L4Env* bietet dazu einen Nameserver (`names`). Bei diesem können sich zum Beispiel Server-Threads mit einem Namen und ihrer Thread-ID registrieren, Client-Threads durch Abfragen des Nameservers die Thread-IDs von Server-Threads ermitteln und somit Dienste von Servern in Anspruch nehmen.

Konsolen dienen der Ein- und Ausgabe von Programmen, das *L4Env* verfügt über verschiedene Konsolen, z.B. `l4con` oder `dope_term`. Des Weiteren gibt es zum Starten von Programmen einen Loader (`loader`) und einen Task-Server (`simple_ts`). Um Zugriff auf Dateien zu ermöglichen bietet das *L4Env* verschiedene Dateiserver. Der `simple_file_server` und der `static_file_server` gehören zum *L4VFS*, das eine POSIX-kompatible Schnittstelle anbietet.

Zurzeit kann der `simple_file_server` nur begrenzt viele Dateien anbieten. Grund dafür ist die zu kleine *Trampoline Page*; diese wird benötigt, um eine Task (Programm) zu starten. Sie enthält den initialen Stack und den Multiboot-Info-Block. Dieser beinhaltet u.a. Module für die Task. Roottask setzt die *Trampoline Page* auf und allokiert standardmäßig nur eine Seite für diese. Da der `simple_file_server` Dateien in Form von Modulen im Multiboot-Info-Block übergeben bekommt, ist die Anzahl der Dateien durch die Größe der *Trampoline Page* begrenzt.

Speziell für L4 Programme stehen die Dateiserver `fuxfprov` und `bmodfs` zur Verfügung. Diese bilden eine Schnittstelle zum Zugriff auf Dateien für L4 Programme an.

Fiasco-UX ist eine Portierung des Fiasco-Mikrokerns auf Linux. Dadurch kann der Fiasco-Mikrokern wie eine normale Anwendung unter Linux gestartet werden. Vorteil des ganzen ist, dass Programme für L4 auf der gleichen Maschine entwickelt und getestet werden können. Weiterhin besteht die Möglichkeit, mehrere Fiasco-UX parallel auf einer Maschine zu Starten.

L4 Programme, gestartet auf Fiasco-UX, können durch `fuxfprov` auf Dateien des darunterliegenden Linux zugreifen. `bmodfs` ermöglicht den Zugriff auf Dateien, die als Module beim Start mitgegeben werden.

Signale werden durch einen Signalserver (`signal_server`) und einer zugehörigen Bibliothek bereitgestellt. Diese bietet eine POSIX-kompatibles Schnittstelle.

1.3 POSIX

POSIX[Posix08] ist eine vom IEEE (Institute of Electrical and Electronics Engineers) entwickelte Sammlung von Standards. Die Entwicklung der Standards begann im Jahr 1988. POSIX (Portable Operating System Interface) beschreibt die Schnittstelle zwischen Programmen und dem Betriebsystem. Der Standard definiert Dienste, die ein Betriebssystem bieten muss, um POSIX kompatibel zu sein.

Die ersten Teile des POSIX-Standards definieren:

- IEEE 1003.1: Basis-Definitionen, Konventionen, bereitzustellende C-Headerdateien
- IEEE 1003.1b: Echtzeiterweiterungen
- IEEE 1003.1c: Threaderweiterungen
- IEEE 1003.2: Shell und Hilfsprogramme

Durch die Definition einer solchen Schnittstelle sind Portierungen POSIX-kompatibler Programme auf POSIX-kompatible Plattformen sehr einfach. POSIX-kompatibel sind z.B. Solaris, VxWorks und UnixWare. Weitgehend POSIX-kompatibel sind beispielsweise GNU/Linux und BSD.

Kapitel 2

L4Python

Aus den Grundlagen des vorherigen Kapitels definiere ich nun Anforderungen an eine Shell für das *L4Env*. Danach beschreibe ich Details der Portierung der Skriptsprache auf L4, d.h. Änderungen an dem Quellcode der Skriptsprache und dessen Buildsystem.

2.1 Design

In diesem Abschnitt vergleiche ich die Anforderungen an eine Skriptsprache für L4 mit den Merkmalen der Skriptsprachen (vgl. Kapitel 1.1) und entscheide damit welche Skriptsprachen für L4 geeignet sind.

2.1.1 Eine Shell für L4

Eine Shell für L4 soll als Werkzeug zum Administrieren und Debuggen von Programmen auf L4 dienen. Das bedeutet, dass Bibliotheksfunktionsaufrufe und Systemaufrufe möglich sein sollen. Weiterhin soll die Shell bzw. der Interpreter über eine Skriptsprache verfügen, damit Abläufe automatisiert werden können. Um Bibliotheksfunktionen ausführen zu können, soll es möglich sein, den Interpreter mit C-Code erweitern zu können. Die benötigten Bibliotheken sollen zur Laufzeit nachgeladen werden, damit die Shell flexibel eingesetzt werden kann. Um Bibliotheken möglichst einfach in die Shell einbinden zu können, sollen notwendige Typumwandlungen zwischen C-Typen und Typen der Shell unterstützt werden. Beispielsweise soll eine Variable des C-Typs `char *` auf eine Variable des Python-Typs `string` abgebildet werden. Dazu benötigt man Funktionen, die entweder per Hand geschrieben oder mit Hilfe eines Werkzeugs generiert werden müssen.

Objektorientierung ermöglicht es komplexe Aufgaben mit der Skriptsprache bzw. Shell durchzuführen, dynamische Typisierung vereinfacht den Umgang mit Variablen.

Je nach dem Umfang der Skriptsprachen benötigen diese die ANSI-C Bibliothek und zum Teil die POSIX-Schnittstelle (siehe Kapitel 1.3). Das *L4Env* bietet viele der POSIX-Funktionen und ermöglicht dadurch die Portierung POSIX-kompatibler Programme.

Ein JIT-Compiler (Just-in-time) kann zur Laufzeit Bytecode in nativen Maschinen-Code übersetzen. Die Ausführung des erzeugten Maschinencodes gegenüber der interpretierten Ausführung des Quellcodes ist schneller und dient der Beschleunigung von Skriptsprachen.

Zur Beobachtung und zur Fehleranalyse von Skripten sind Debugger und Möglichkeiten zur Kontrollflussverfolgung geeignet. Ruby, Perl und Python bringen einen eingebauten Debugger mit, weiterhin bieten sie die Möglichkeit der Kontrollflussverfolgung von Skripten([LangStudy08]).

Alle genannten Shells bringen die benötigten Voraussetzungen für eine Portierung auf POSIX-kompatible Systeme mit. Bezüglich des Funktionsumfangs der Skriptsprachen dominieren Ruby, Perl und Python. Die Bash eignet sich nicht als Shell für L4, da sie über keinen direkten Mechanismus zur Erweiterung mit Bibliotheken bzw. C-Code verfügt und nur über eine primitive

	Bash	**Ruby**	**Lua**	**Perl**	**Python**
Interpreter	X	X	(X)	X	X
JIT-Compiler		X	X	X	X
Skriptsprache	X	X	X	X	X
Erweiterbar mit C-Code		X	X	X	X
Einbettbar in C-Code		X	X	X	X
Plattform	Linux, Windows, Mac OS X	POSIX-kompatible Systeme	unabhängig, benötigt C-Bibliothek	POSIX-kompatible Systeme	POSIX-kompatible Systeme
Objektorientiert		X	X	X	X
dynamische Typisierung	X		X	X	X
eingebauter Debugger		X		X	X
Kontrollflussverfolgung (execution tracer)	X	X		X	X

Tabelle 2.1: Vergleich Shells

Skriptsprache mitbringt. Da Lua in erster Linie zum Einbetten in Programme entwickelt ist, eignet es sich auch nur bedingt als Shell für L4.

Ruby, Perl und Python besitzen alle notwendigen Eigenschaften und Funktionalitäten einer Shell für L4. Sie erfüllen alle Voraussetzungen für eine Portierung und Nutzung unter L4. Rein subjektiv betrachtet ist Quellcode von Perl schlechter zu lesen und zu warten als von Python oder Ruby.

Python ist eine leicht zu erlernende Sprache. Da ich gute Erfahrungen mit dieser Sprache gemacht habe, entschied ich mich Python auf L4 zu portieren. Im nächsten Abschnitt stelle ich Python kurz vor und erläutere die Erweiterungsmöglichkeiten der Skriptsprache.

2.1.2 Python

Python ist eine mächtige Skriptsprache, die viele Programmierparadigmen (Objektorientierte-, Aspektorientierte-, Funktionale-Programmierung) ermöglicht und über viele Bibliotheken verfügt. Der interaktive Interpreter macht Python zu einer vollwertigen Shell. Da Pythonskripte nicht direkt auf der Maschine ausgeführt, sondern interpretiert werden, sind diese plattformunabhängig.

Grundlegende Module und Funktionalitäten (u.a. Listen/Mengen/Tupel, Sequenzen) sind in Python fest eingebaut. Erweitern lässt sich Python durch dynamisches Nachladen von *extension modules*. Dies sind Bibliotheken die in Python oder in C geschrieben sein können.

In Python geschriebene *extension modules* lädt Python selbst. Durch das Übersetzen von in Python geschriebenen *extension modules* in Bytecode kann das Laden und Ausführen von *extension modules* erheblich beschleunigt werden.

Das Laden der in C geschriebenen *extension modules* geschieht durch den *dynamic linking loader*. Alternativ können *extension modules* auch statisch in das Python-Binary gelinkt werden. Dadurch kann das Fehlen des *dynamic linking loader* kompensiert bzw. das explizite Bereitstellen der *extension modules* vermieden werden.

Python ist für viele Plattformen verfügbar. Es benötigt grundsätzlich eine C-Bibliothek und die POSIX-Schnittstelle um übersetzt zu werden.

2.2 Implementation

In diesem Abschnitt stelle ich ein paar Details der Portierung von Python auf L4 vor.

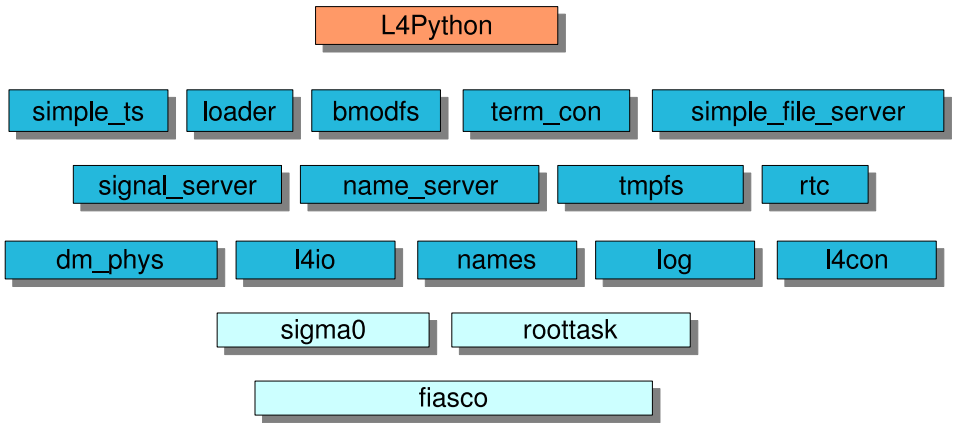

Abbildung 2.1: Python zur Laufzeit

L4Python ist eine Portierung von Python 2.5.1 auf L4. Die Portierung nutzt das *L4Env* als Schnittstelle zum Betriebssystem.

Zum Erstellen von Python verwendet meine Portierung das Buildsystem von Python selbst. Da Python das GNU build system, also *autoconf* und *automake* benutzt, beschränken sich sich die Änderungen im wesentlichen auf die Dateien `configure.in` und `Makefile.pre.in`. Geändert werden mussten Flags und Bibliotheksvariablen (z.B. `LDFLAGS` , `CPPFLAGS`). *Extension module*, die besondere Bibliotheken benötigen oder nicht nutzbare Funktionalität bieten habe ich deaktiviert. Das betrifft u.a. die *extension modules* für *curses*, Sockets, SSL, NIS, zlib und Audio-Unterstützung. Python benutzt sich selbst, um einige Module zu erzeugen. Da L4Python unter Linux gebaut wird, entsteht das Problem, dass ein Linux-Python für die Erstellung von L4Python benötigt wird. Im Detail wird ein Python-Interpreter (`python`) und ein Python-Parsergenerator (`pgen`) benötigt. Da der Python-Parsergenerator meist nicht in den Linux-Distribution enthalten ist und um Probleme unterschiedlicher Python-Version zu umgehen, empfiehlt es sich, ein Linux-Python 2.5.1 zu bauen und zu verwenden.

Mit Hilfe von Dummys habe ich fehlende Funktionen abgedeckt.

Für die Aus- und Eingabe benötigt Python eine Konsole, die Portierung L4Python unterstützt `l4con` und DOpE (`dope_term`). Um eine Konsole zu nutzen müssen die drei Standard I/O-Streams (`stdin`, `stdout`, `stderr`) mit dieser assoziiert sein. Diese Assoziation wird beim Start von Python durch die Funktionen `init_dope_term` bzw. `init_con_term` durchgeführt. Die Implementation der Funktionen zeigt Abbildung 2.2.

L4Python bildet Python-Threads auf native L4 Threads ab; dazu wird die Threadbibliothek (`thread`) benutzt. Als Lockprimitive verwendet L4Python Semaphore, die durch die Semaphorenbibliothek (`semaphore`) bereitgestellt werden. Das Implementieren der in Abbildung 2.3 genannten Funktionen war relativ einfach, meist konnten die korrespondierenden Funktionen aus der Thread- bzw. Semaphorenbibliothek von L4 aufgerufen werden.

Python benötigt POSIX-Funktionalität; diese wird durch das *L4VFS* bereitgestellt:

- Python benutzt zum dynamischen Laden von Bibliotheken die Funktionen `dlopen()`, `dlerror()`, `dlsym()` und `dlclose()`; diese werden vom *dynamic linking loader* bereitgestellt. Der *dynamic linking loader* von L4 befindet sich im Paket *ldso* und stellt diese Funktionen zur Verfügung.

- Als Dateiserver dienen `simple_file_server` und `bmodfs` bzw. `fuxprov`.

- Python benötigt POSIX-Signale, diese werden von dem Signal-Server (`signal_server`) zur Verfügung stellt.
- `fstab` generiert einen Namensraum für das *L4VFS* und bindet Dateiserver in diesen ein.

```
void init_dope_term(void) {
    [...]
    l4vfs_attach_namespace(ns, VC_SERVER_VOLUME_ID, "/", "/linux")
    [...]
    fd = open("/linux/vc0", O_RDONLY);
    fd = open("/linux/vc0", O_WRONLY);
    fd = open("/linux/vc0", O_WRONLY);
    [...]
}
void init_con_term(void) {
    [...]
    l4vfs_attach_namespace(ns, TERM_CON_VOLUME_ID, "/", "/linux"))
    [...]
    fd = open("/linux/vc0", O_RDONLY);
    fd = open("/linux/vc0", O_WRONLY);
    fd = open("/linux/vc0", O_WRONLY);
    [...]
}
```

Abbildung 2.2: Implementation der Funktionen zur Initialisierung der Konsolen

```
static void PyThread__init_thread(void);
/* Thread support. */
long PyThread_start_new_thread(void (*func)(void *), void *arg);
long PyThread_get_thread_ident(void);
static void do_PyThread_exit_thread(int no_cleanup);
void PyThread_exit_thread(void);
void PyThread__exit_thread(void);

#ifndef NO_EXIT_PROG
static void do_PyThread_exit_prog(int status, int no_cleanup);
void PyThread_exit_prog(int status);
void PyThread__exit_prog(int status);
#endif /* NO_EXIT_PROG */
/*  Lock support. */
PyThread_type_lock;
PyThread_allocate_lock(void);
void PyThread_free_lock(PyThread_type_lock lock);
int PyThread_acquire_lock(PyThread_type_lock lock, int waitflag);
void PyThread_release_lock(PyThread_type_lock lock);
```

Abbildung 2.3: Funktionsprototypen für Threads und Semaphore in Python

Kapitel 3

Erweitern von L4Python

Aufbauend auf der Portierung von Python auf L4 aus Kapitel 2 möchte ich mich nun mit der Möglichkeit beschäftigen, den Python-Interpreter zu erweitern. Dazu werde ich Pythons Mechanismus für Erweiterungen vorstellen und Möglichkeiten der Integration von L4 Bibliotheken diskutieren.

3.1 Analyse

Pythons Mechanismus für Erweiterungen sind *extension modules*. Diese ermöglichen das Einbinden von Bibliotheken in Python. In diesem Abschnitt möchte ich beschreiben, wie *extension modules* erstellt werden und was bei der Integration von L4 Bibliotheken beachtet werden muss.

3.1.1 Erweitern des Python-Interpreters mit C

Der Python-Interpreter kann durch *extension modules* um Funktionen, geschrieben in C, erweitert werden. Diese Module können Funktionen, Variablen, Exceptions und neue Typen beinhalten und Bibliotheksfunktionen oder Systemaufrufe tätigen. Zur Implementation dieser Module bietet Python Funktionen, Makros und Variablen zum Zugriff auf die Laufzeitumgebung an.

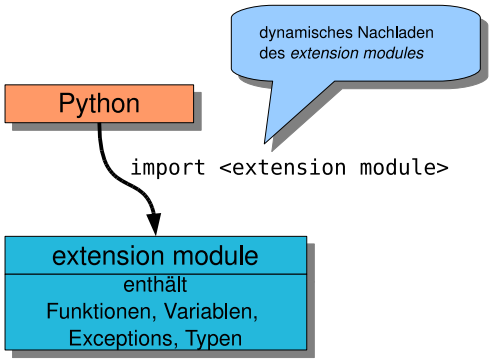

Abbildung 3.1: Nachladen von dynamisch gelinkten *extension modules*

Extension modules können zur Laufzeit geladen werden (siehe Abbildung 3.1) oder statisch in den Python-Interpreter hineingelinkt werden (siehe Abbildung 3.2). Der Vorteil beim Laden zur Laufzeit besteht in der sparsamen Speichernutzung durch einen kleinen Interpreter, der benötigte Module nachlädt. Durch statisches Hineinlinken der *extension modules* in den Python-Interpreter lassen sich diese schneller laden bzw. müssen nicht explizit bereitgestellt werden.

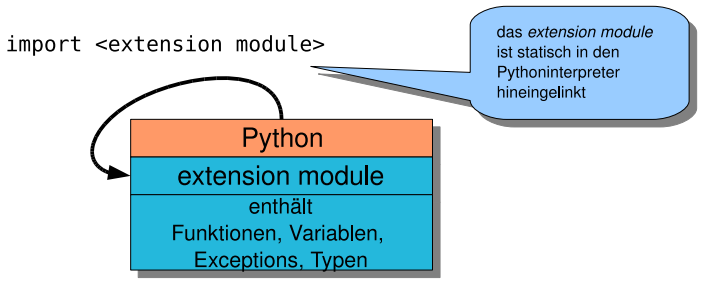

Abbildung 3.2: Nachladen von statisch in den Python-Interpreter gelinkten *extension modules*

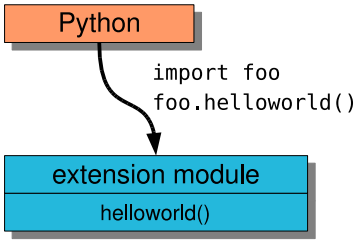

Abbildung 3.3: Ausführen von Funktionen aus *extension modules*

Zum Ausführen von Funktionen aus *extension modules* müssen diese importiert werden, danach können auf alle Funktionen, Variablen usw. des *extension module* zugegriffen werden(siehe Abbildung 3.3).

Die Generierung der *extension modules* kann per Hand oder mit Hilfe eines Werkzeugs erfolgen. Im Abschnitt „Design und Implementation“ (3.2) werde ich beide Varianten vorstellen.

3.1.2 Integration von L4 Bibliotheken in Python

Um Administrations- und Debugging-Aufgaben durch Pythonskripte realisieren zu können, muss mit anderen L4 Anwendungen kommuniziert werden. Die Kommunikation erfolgt mittels Bibliotheksfunktion, welche durch IPC mit L4 Anwendungen kommunizieren.

Zur Kommunikation aus Python heraus mit L4 Anwendungen müssen Funktionen aus L4 Bibliotheken aufgerufen werden. Dazu sind Konvertierungen für die Aufruf- und Rückgabe- Parameter der Bibliotheks-Funktionsaufrufe erforderlich. Die Konvertierungen (Typumwandlungen) und der Aufruf der Bibliotheksfunktionen erfolgt in einem *extension module*.

Um die Bibliotheksfunktionen aus dem *extension module* heraus aufrufen zu können, müssen die L4 Bibliotheken **statisch** zum *extension module* gelinkt oder **dynamisch** zur Laufzeit nachgeladen werden.

Bei beiden Ansätzen muss das *extension module* nach dem Start von Python geladen werden. Beim statischen Ansatz können danach L4 Bibliotheksfunktionen sofort ausgeführt werden, währenddessen beim dynamischen die benötigten L4 Bibliotheken noch nachgeladen werden müssen.

Um neue Bibliotheken in Python verfügbar zu machen, müssen diese beim dynamischen Ansatz einfach nachgeladen werden. Das *extension module* muss nicht neu übersetzt werden. Beim statischen Ansatz hingegen muss das *extension module* neu übersetzt und die Bibliotheken dazugelinkt werden.

3.2 Design und Implementation

Das Erstellen von *extension modules* kann von Hand oder mit Hilfe eines Werkzeugs erfolgen. Im Folgenden möchte ich kurz die beiden Möglichkeiten vorstellen und Vor- bzw. Nachteile ansprechen. Im darauffolgenden Abschnitt diskutiere ich das Zusammenspiel von *extension modules* und L4 Bibliotheken.

3.2.1 Generieren der *extension modules* von Hand

Das Erstellen von *extension modules* von Hand macht es möglich, Bibliotheksfunktion aus Python heraus aufzurufen. Das Ziel ist die Funktion `useful_function()` aus einer Bibliothek auszuführen.

```
int useful_function(int x) {
/*code that does something useful*/
}
```

Ein *extension module* enthält Wrapper für Bibliotheksfunktionen und Initialisierungscode für Python. Die Funktion `useful_function` wird durch die Wrapperfunktion `wrap_useful_function` ausgeführt; vor und nach der Ausführung müssen Typumwandlungen von Python in C-Typen bzw. umgekehrt implementiert werden.

```
#include <Python.h>
PyObject *wrap_useful_function(PyObject *self, PyObject *args) {
    [...]
    result = useful_function(x);
    [...]
}
static PyMethodDef exampleMethods[] = {
    {"useful_function", wrap_useful_function, METH_VARARGS},
    {NULL, NULL}
};
void initialize_function(){
    Py_InitModule("exampleModule", exampleMethods);
}
```

Für das Ausführen der Funktionen aus Python heraus muss das entsprechende *extension module* importiert werden. Danach kann die Funktion einfach ausgeführt werden.

```
import exampleModule
ret = exampleModule.useful_function(10)
```

Da die Wrapperfunktionen selbst implementiert werden müssen; bietet die händische Erstellung von *extension modules* maximale Flexibilität. In Wrapperfunktionen können mehrere Funktionen ausgeführt werden. Dies bietet viele Vorteile: Beispielsweise können komplexe Vor- und Nachbedingen für den Zielfunktionsaufruf implementiert werden.

Da der Aufwand bei der händischen Erstellung von *extension modules* erheblich ist, wäre eine automatische Generierung von *extension modules* wünschenswert. Im nächsten Abschnitt stelle ich einige Werkzeuge vor, mit denen *extension modules* generiert werden können.

3.2.2 Generieren der *extension modules* mit Hilfe eine Werkzeugs

Extension modules, welche Wrapper für Bibliotheken bilden, enthalten Code für Typumwandlungen und den Bibliotheksfunktionsaufrufe. Die Generierung dieser Wrapper bzw. der *extension module* kann sehr gut mit Werkzeugen automatisiert werden.

Solche Werkzeuge generieren die Typumwandlungs-Routinen und die Wrapperfunktionen automatisch. Für Python gibt es viele solcher Werkzeuge; manche sind nur für C++ geeignet, andere bringen komplexe Laufzeitumgebungen mit. Diese Laufzeitumgebungen sind meist *extension modules* und enthalten Code für Typumwandlungen zwischen Python- und C-Typen.

SIP [SIP08] ist ein Werkzeug, das Python-Bindings für C und C++ Bibliotheken generiert. SIP wurde ursprünglich 1998 für PyQt entwickelt. PyQt sind Python-Bindings für das Qt GUI toolkit. SIP konvertiert automatisch die Standardtypen von Python bzw. C/C++. Es unterstützt viele C++ Sprachmerkmale und bringt ein Buildsystem für die generierten *extension modules* mit.

SWIG [SWIG08] ist eine Softwareentwicklungswerkzeug, das in C oder C++ geschriebenen Code in vielen Hochsprachen verfügbar macht. Es wurde ursprünglich in der Theoretical Physics Division am Los Alamos National Laboratory entwickelt. Es war zum Generieren von Schnittstellen für wissenschaftliche Simulationen, die auf der Connection Machine 5 liefen, gedacht. Es kann nicht nur Python mit C-Code erweitern, sondern ebenso viele andere Skriptsprachen wie z.B. Lua, Java, und PHP.

PyCXX [PyCXX08] ist eine Menge von Klassen, mit deren Hilfe *extension modules* für Python gebaut werden können. Diese Klassen kapseln die Python C API, fangen Ausnahmen (exceptions) und kümmern sich um die Speicherverwaltung in Python (die Speicherverwaltung Pythons basiert auf einem sogenannten Reference-Counting-Algorithmus – Instanzen von Objekten werden zur Entsorgung freigegeben, sobald keine Referenzen mehr auf sie bestehen).

Boost.Python [Boost08] ist eine C++ Bibliothek, die eine Zusammenarbeit von Python und C++ ermöglicht. Sie kann viele Typen automatisch umwandeln und verfügt über ein automatisches Buildsystem für die *extension modules*.

Pyrex [Pyrex08] ist eine Sprache, um Python *extension modules* zu schreiben. Man kann Pyrex als Python mit C-Datentypen bezeichnen; es unterstützt das Schreiben der *extension modules* bei Typumwandlungen, Fehlerbehandlungen und der Speicherverwaltung von Python.

In meiner Arbeit habe ich SIP und SWIG benutzt, um L4 Bibliotheken in Python verfügbar zu machen. Dazu musste bei SIP die Laufzeitbibliothek portiert und das Buildsystem angepasst werden. Die SIP-Laufzeitumgebung enthält u.a. Methoden für Typumwandlungen. SWIG generiert lediglich den Quellcode der *extension modules*; daher war keine Portierung von SWIG notwendig.

Die anderen genannten Werkzeuge eignen sich nur bedingt zur Integration von L4 Bibliotheken in Python. PyCXX und Boost.Python unterstützten nur C++; Pyrex befindet sich noch in Entwicklung.

Im nächsten Abschnitt beschäftige ich mich mit Möglichkeiten L4 Bibliotheken in Python zu nutzen. Dazu werden zwei genannten Varianten des Erstellens von *extension modules* – per Hand oder mit Hilfe eines Werkzeugs – genutzt.

3.3 Extension modules und L4 Bibliotheken

Dieser Abschnitt zeigt implementationstechnische Details der im Abschnitt „Analyse“ (3.1) diskutierten Möglichkeit, Python durch Bibliotheken zu erweitern. Im Folgenden werde ich die zwei Ansätze zum Ausführen von L4 Bibliotheksfunktionen aus Kapitel „Integration von L4 Bibliotheken in Python“ (3.1.2) diskutieren.

Beim **dynamischen** Ansatz werden die benötigten Bibliotheken mittels eines *extension module* nachgeladen und ausgeführt (siehe Abbildung 3.4).

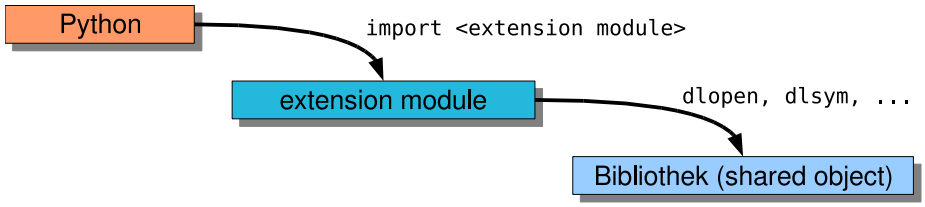

Abbildung 3.4: der dynamische Ansatz

Beim **statischen** Ansatz werden die Bibliotheken statisch in das *extension module* gelinkt. Durch Laden/Importieren dieser *extension modules* ist das Ausführen der Funktionen dann in Python möglich (siehe Abbildung 3.5).

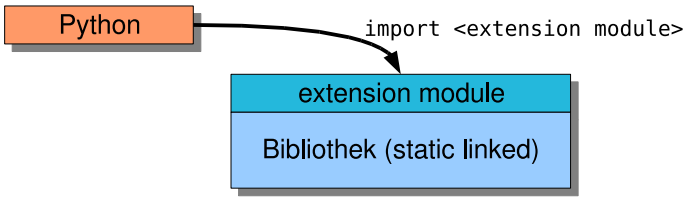

Abbildung 3.5: der statische Ansatz

Bibliotheken und *extension modules* können sowohl dynamisch nachgeladen sowie statisch gelinkt werden. In Abbildung 3.6 sind alle möglichen Kombination von *extension modules* und Bibliotheken dargestellt. Bei Kombination (a) wird sowohl das *extension module* (von Python selbst) sowie die Bibliothek (vom *extension module*) dynamisch nachgeladen. Bei (b) ist die Bibliothek statisch in das *extension module* gelinkt, welches dynamisch nachgeladen wird. Bei (c) ist das *extension module* statisch in das Python-Binary gelinkt. Die Bibliothek wird durch das *extension module* dynamisch nachgeladen. Bei (d) ist sowohl das *extension module* als auch die Bibliothek statisch in das Python-Binary gelinkt.

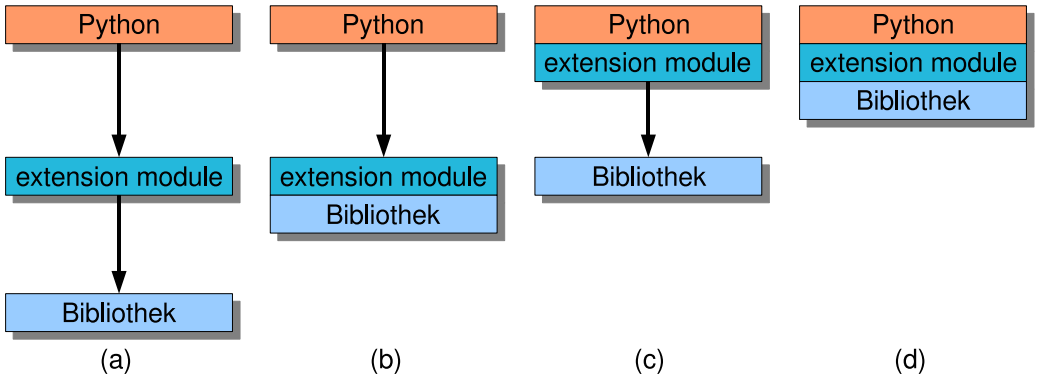

Abbildung 3.6: *extension modules* und Bibliotheken

3.3.1 Dynamisches Nachladen von L4 Bibliotheken durch ein *extension module*

Um Funktionen auszuführen, muss die Eingabe (Funktionsname und Argumente) des Benutzers mittels eines Parsers analysiert werden. Danach muss die erforderliche, zu ladende Bibliothek ermittelt werden. Dies kann mit Hilfe einer Datenbank, die zu jeder Funktion die entsprechende Bibliothek kennt, geschehen. Eine andere Möglichkeit ist eine Heuristik, die anhand der eingegebenen Funktionsnamen die Bibliothek ermittelt. Die benötigten Bibliotheken werden dann nachgeladen und die entsprechende Funktion ausgeführt (siehe Abbildung 3.7).

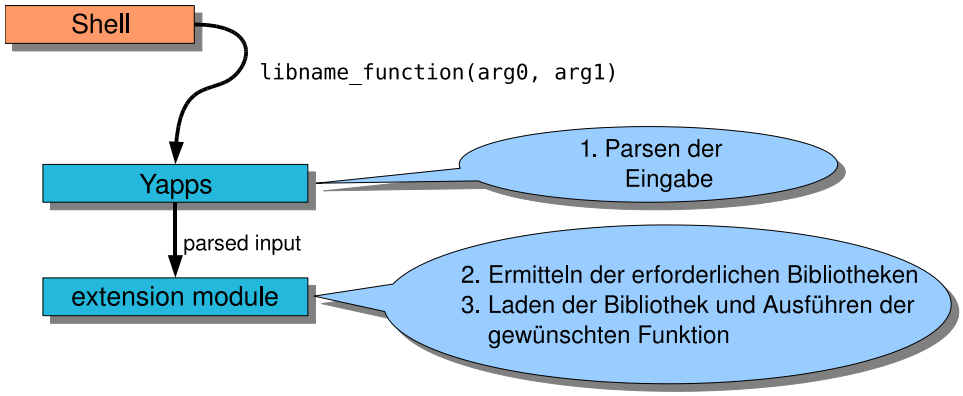

Abbildung 3.7: Parsen und Ausführen der gewünschten Funktion

Meine Implementation des dynamischen Ansatzes benötigt einen Parser, der C-Funktionsaufrufe analysieren kann, da Funktionen aus C-Bibliotheken aufgerufen werden sollen.

Yapps (Yet Another Python Parser System)[Yapps08] ist ein Parsergenerator, welcher basierend auf einer gegebenen Grammatik Parser generieren kann. Yapps ist in Python geschrieben und kann daher sehr leicht in Pythonskripte eingebettet werden. Notwendig ist lediglich eine Spezifikationsdatei, welche eine Grammatik enthält. Daraus generiert Yapps einen Parser; da dieser generierte Parser in Python geschrieben ist, war eine Portierung von Yapps auf L4 nicht notwendig.

Nachdem die Eingabe des Benutzers analysiert wurde, müssen die benötigten Bibliotheken ermittelt werden. Die geschieht in meiner Implementation durch eine Heuristik, welche den Präfix des Funktionsnamens als Bibliotheksnamen interpretiert.

Die gesamte Implementation vom Parsen der Eingabe bis zum Ausführen einer Bibliotheks-Funktion habe ich mit Hilfe eines *extension modules* realisiert.

Der dynamische Ansatz ist sehr flexibel und erweiterbar. Da neue Bibliotheken keine Änderung meiner Implementation bzw. keine Neu-Übersetzung des *extension modules* erfordern, sollten diese sofort nutzbar sein. Einzige Voraussetzung ist, dass benötigte Typumwandlungs-Routinen existieren.

Nachteil ist der Aufwand der Implementation von Typumwandlungs-Routinen. Diese müssen komplett per Hand geschrieben werden. Weiterhin ist es nicht möglich, die Argumenttypen zu verifizieren; auch die Anzahl der benötigten Funktionsargumente ist unbekannt. Diese Informationen könnten aus den Headerdateien der Bibliotheken während der Übersetzung geholt und zur Laufzeit benutzt werden – dadurch würde jedoch die Flexibilität bzgl. neuer Bibliotheken wegfallen.

Insgesamt ist der dynamische Ansatz mit viel Aufwand realisierbar – weit weniger Aufwand und mehr Möglichkeiten zur Automatisierung der Implementation bietet der statische Ansatz, welcher im folgenden vorgestellt wird.

3.3.2 Statisches Linken der Bibliotheken in ein *extension module*

Im Gegensatz zum dynamischen Ansatz aus dem vorangegangenden Kapitel, werden beim statischen Ansatz die Bibliotheken statisch in ein *extension module* hineingelinkt. Dies hat den Vorteil das nur ein *extension module* geladen werden muss, weiteres nachladen von Bibliotheken, wie beim dynamischen Ansatz, entfällt. Im folgenden zeige ich wie der statische Ansatz per Hand oder mit Hilfe eines Werkzeugs umgesetzt werden kann.

Erstellen der *extension modules* per Hand
Die Implementation von Wrappern für die Bibliotheksfunktionen erfolgt in einem *extension module*. Bibliotheken werden statisch in das *extension module* gelinkt. Die Wrapper bestehen aus notwendigen Typumwandlungen für die Argumente, dem Aufruf der gewünschten Bibliotheksfunktion und den Typumwandlungen für die Rückgabewerte.

Dieser Ansatz ist flexibel, aber relativ aufwendig. Alle Typumwandlungen von Python nach C/C++ und umgekehrt müssen per Hand implementiert werden. Vorteilhaft ist, dass Vor- und Nachbedingungen implementiert werden können; dadurch können Ausnahmen (Exceptions) beim Aufruf der Bibliotheksfunktionen abgefangen werden.

Generieren von *extension modules* mittels eines Werkzeugs
SIP ermöglicht auf einfache Art und Weise, Bibliotheksfunktionen aus Python heraus aufzurufen. Dazu muss per Hand eine Spezifikationsdatei geschrieben werden; diese beinhaltet im Wesentlichen die Prototypen der Bibliotheksfunktionen, die in Python verfügbar sein sollen. Notwendige Typkonvertierungen werden von SIP weitgehend automatisch durchgeführt.

```
 %CModule sip_example 0
/* function prototypes, copy & paste from header file */
int useful_function(int x);
```

Das Generieren der *extension modules* sowie das Übersetzen dieser erfolgt durch das SIP Buildsystem. Die von SIP generierten *extension modules* enthalten Wrapperfunktionen für die Bibliotheksfunktionen sowie notwendige Typumwandlungs-Routinen.

```
[...]
static PyObject *func_useful_function(PyObject *sipSelf,PyObject *sipArgs)
{
    int sipArgsParsed = 0;
    {
        int a0;
        if (sipParseArgs(&sipArgsParsed,sipArgs,"i",&a0))
        {
```

```
            int sipRes;
            sipRes = useful_function(a0);
            return PyInt_FromLong(sipRes);
        }
    }
    /* Raise an exception if the arguments couldn't be parsed. */
    sipNoFunction(sipArgsParsed,sipNm_sip_sip_example_useful_function);
    return NULL;
}
[...]
```

Das Ausführen von den Funktionen in Python erfolgt durch Importieren der *extension modules* und Aufrufen der gewünschten Funktion.

```
import sip_example
ret = sip_example.useful_function(10)
```

SIP bringt Mechanismen zur automatischen und manuellen Typumwandlung zwischen C, C++ Typen und Python Typen mit. Für viele Standardtypen, wie zum Beispiel Zeichenketten/Strings (`char*`) und Zahlen (`int, double`), verfügt SIP über fertige Routinen zur Typumwandlung mit.

Der einfachste und flexibelste Mechanismus von SIP ermöglichst eine eigene Implementation der Wrapperfunktionen in der Spezifikationsdatei. Dadurch können Typumwandlungen frei definiert und so auch komplexe Typen auf Pythontypen abgebildet werden. Es besteht zusätzlich zur Definition der Typumwandlungen die Möglichkeit, Vor- und Nachbedingungen zu implementieren.

```
int function(type arg);
%MethodCode
    [...]
    function(...)
    [...]
    sipRes =
%End
```

Der Aufwand bei der vollständigen Implementation von Wrapperfunktionen ist relativ hoch. SIP bietet die Möglichkeit Routinen zur Typumwandlung anzugeben; diese werden dann bei der Generierung des Wrappercodes benutzt.

```
%MappedType type
{
    %TypeHeaderCode
         // handwritten code that specify the library interface to
            the type being mapped
    %End
    %ConvertToTypeCode
         // handwritten code that converts a Python object to
            a C type instance
    %End
    %ConvertFromTypeCode
         // handwritten code that converts an instance of
            a C type to a Python object
    %End
}
```

SWIG ermöglicht es, ähnlich wie SIP, Bibliotheksfunktionen aus Python heraus aufzurufen. Die Spezifikationsdatei enthält wie SIP Funktionsprototypen und Typumwandlungs-Routinen.

```
%module swig_example
/* function prototypes, copy & paste from header file */
extern int useful_function(int x);
```

Vor dem Aufrufen der Funktionen muss das von SWIG erstellte *extension module* importiert werden. Danach können die Bibliotheksfunktionen aus Python heraus aufgerufen werden.

```
import swig_example
ret = swig_example.useful_function(10)
```

Wie SIP verfügt auch SWIG über erweiterte Möglichkeiten, Typumwandlungen zu definieren.

```
%CModule sip_example 0
int my_function(...) {
    [...]
    function(...);
    [...]
}
%}
%name(function) int my_function(...);
```

3.3.3 Beispiel: Names

`Names` ist ein Nameserver für L4. Er wird benötigt, damit Threads per IPC miteinander kommunizieren können. Er bietet Möglichkeiten andere Threads zu Finden bzw. neue Threads zum Registrieren.

Um die Bibliotheksfunktionen von Names in Python verfügbar zu machen, habe ich mit Hilfe von SIP ein *extension module* erstellt. SIP benötigt eine Spezifikationsdatei, die Prototypen der Bibliotheksfunktionen, welche in Python verfügbar sein sollen, enthält. Zur Ausgabe bzw. Erzeugung von Thread-IDs (`l4threadid_t`) in Python habe ich Hilfsfunktionen `l4threadidt_to_list`, `l4threadid_to_str` und `build_l4threadid` implementiert.

Abbildung 3.8 enthält notwendig Typumwandlungen, Prototypen der Client-Bibliothek von Names und Hilfsfunktionen zur Behandlung von Thread-IDs.

Um die Funktionen in Python ausführen zu können, muss als erstes das *extension module* `sip_names` importiert werden. Danach können die Bibliotheksfunktionen von Names durch Voranstellen von `sip_names.` ausgeführt werden.

```
import sip_names

print "My thread id is %d" % (sip_names.l4thread_myself())
sip_names.names_dump()

(ret, id) = sip_names.names_query_name("loader.pager")
print "Thread id from loader.pager is", sip_names.l4threadidt_to_str(id)
print "[raw, version_low, lthread, task, version_high] =",
                    sip_names.l4threadidt_to_list(id)

sip_names.names_register("newthread")

(ret, id) = sip_names.names_query_name("newthread")
print "Thread id from newthread is", sip_names.l4threadidt_to_str(id)

newid = sip_names.build_l4threadidt(0,5,20,0)
sip_names.names_register_thread_weak("newthread", newid)

(ret, id) = sip_names.names_query_name("newthread")
print "Thread id from newthread is", sip_names.l4threadidt_to_str(id)
```

Das Beispiel demonstriert das Abfragen des Nameservers `sip_names.names_query_name(...)` und das Registrieren `sip_names.names_register(...)` von Threads.

Mit Hilfe von `sip_names.l4threadidt_to_list(...)` kann auf die einzelnen Komponenten einer Thread-ID zugegriffen werden, während `sip_names.l4threadidt_to_str(...)` einen String aus einer Thread-ID erzeugt.

Zum Generieren von Thread-IDs dient die Funktion `sip_names.build_l4threadidt(...)`. Diese benötigt vier Argumente – `version_low`, `lthread` , `task` und `version_high` – um eine Thread-ID zu generieren.

```
%CModule sip_names 0
%ModuleHeaderCode
#include <l4/util/l4_macros.h>
#include <l4/sys/types.h>
%End
/* simple type mapping */
typedef int l4thread_t;
typedef int l4_threadid_t;
/* function prototypes, copy & paste from header file */
l4thread_t l4thread_myself();
int names_register(const char* name);
int names_register_thread_weak(const char* name, struct l4_threadid_t id);
int names_unregister(const char* name);
int names_unregister_thread(const char* name, l4_threadid_t id);
int names_query_name(const char* name, l4_threadid_t* id);
int names_query_id(const l4_threadid_t id, char* name, const int length);
int names_waitfor_name(const char* name, l4_threadid_t* id, const int timeout);
int names_query_nr(int nr, char* name, int length, l4_threadid_t *id);
int names_unregister_task(l4_threadid_t tid);
int names_dump();
char *l4threadidt_to_str(int);
%MethodCode
    char *str = malloc(sizeof(char) *  16);
    l4_threadid_t *id = (l4_threadid_t *) &a0;
    sprintf(str, l4util_idfmt"", l4util_idstr(*id));
    sipRes = str;
%End
void l4threadidt_to_list(int, int*, int*, int*, int*, int*);
%MethodCode
    l4_threadid_t *id = (l4_threadid_t *) &a0;
    PyObject *list = PyList_New((Py_ssize_t)0);
    a1 = (*id).raw;
    a2 = (*id).id.version_low;
    a3 = (*id).id.lthread;
    a4 = (*id).id.task;
    a5 = (*id).id.version_high;
%End
int build_l4threadidt(int, int, int, int);
%MethodCode
    l4_threadid_t *id = malloc(sizeof(l4_threadid_t));
    int *ret = id;
    (*id).id.version_low = a0;
    (*id).id.lthread = a1;
    (*id).id.task = a2;
    (*id).id.version_high = a3;
    sipRes = *ret;
%End
```

Abbildung 3.8: Spezifikationsdatei für Names

3.3.4 Beispiel: L4sys

L4sys bildet das ABI (application binary interface) zum L4 Mikrokernel. Mit Hilfe von *L4sys* ist es möglich, Dienste zu implementieren, da es Systemaufrufe und Interprozesskommunikation (IPC) ermöglicht.

Um Dienste in Python zu implementieren, habe ich die IPC-Funktionen zum Senden und Empfangen in L4Python verfügbar gemacht. Dazu habe ich ein *extension module* per Hand implementiert. Es gibt unterschiedliche IPC-Varianten; als erstes habe ich der Short-IPC in Python verfügbar gemacht. Mit Short-IPC können 2 Wörter (32 Bit) zu einem anderen Thread gesendet werden. Die zweite IPC-Variante, die ich für L4Python verfügbar gemacht habe, ist Long-IPC. Damit ist es möglich, Strings (char *) und Wörter zu verschicken.

Ich habe das *extension module* für IPC per Hand, da kaum Möglichkeiten zur Automatisierung mit Hilfe eines Werkzeugs in diesem Fall existieren.

Das *extension module* stellt für jede IPC-Variante zwei Funktionen zur Verfügung:

- `[ret, rcv_dword0, rcv_dword1] l4_short_ipc_receive(threadid)` und
- `[ret] l4_short_ipc_send(threadid, snd_dword0, snd_dword1)` für Short-IPC bzw.
- `[ret, rcv_dword0, rcv_dword1, rcv_str] l4_long_ipc_receive(threadid)` und
- `[ret] l4_long_ipc_send(threadid, snd_dword0, snd_dword1, snd_str)` für Long-IPC.

Abbildung 3.9 zeigt die Implementation der Funktionen zum Senden und Empfangen im Fall Short-IPC. Als Erstes werden die Argumente geparst; Python bietet dazu die C-Funktion `Py_Arg_ParseTuple(...)` an. Danach werden alle notwendigen Variablen entsprechend der geparsten Argumente gesetzt. Nun kann eine IPC gesendet oder empfangen werden. Dies geschieht durch den Funktionsaufruf `l4_ipc_send(...)` bzw. `l4_ipc_receive(...)`. Abschließend gibt die Funktion den Status des Sendens bzw. Empfangens und eventuell weitere Werte zurück.

Python serialisiert standardmäßig alle Funktionsaufrufe eines *extension modules*. Damit trotzdem mehrere Threads parallel Funktionen aus einem *extension module* ausführen können, stellt Python das Makro `Py_BEGIN_ALLOW_THREADS` bzw. `Py_END_ALLOW_THREADS` zur Verfügung.

Wenn zwei Pythonthreads per IPC kommunizieren wollen, tritt genau dieser Fall ein. Damit die Komunikation per IPC zwischen Pythonthreads möglich ist, habe ich die genannten Makros am Anfang und am Ende der Sende- bzw. Empfangs-Funktionen eingefügt.

Das Senden und Empfangen von IPCs ist damit sehr einfach in Python möglich:

```
import sys, l4sys, sip_names, threading, time, thread

class IPCReceiveThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        sip_names.names_register("receive");
        (self.ret, self.id) = sip_names.names_query_name("send")
        rcv = l4sys.l4_long_ipc_receive(self.id)
        print rcv

rt = IPCReceiveThread() rt.start()
sip_names.names_register("send")
(ret, id) = sip_names.names_query_name("receive")
l4sys.l4_long_ipc_send(id,"helloworld",10,20)
```

Beide Threads registrieren sich beim Nameserver (`sip_names.names_register(...)`) und ermitteln dann jeweils die Thread-ID des anderen (`sip_names.names_query_name(...)`). Danach sendet der eine Thread dem anderen eine IPC (`l4sys.l4_long_ipc_send/receive(...)`).

```
PyObject *call_l4_long_ipc_send(PyObject * self, PyObject * args)
{
    [...]

    Py_BEGIN_ALLOW_THREADS
    if (!PyArg_ParseTuple(args, "is#ii", &threadid, &_snd_str,
                                         &len, &_snd_dword0, &_snd_dword1)) {
        PyErr_SetString(PyExc_TypeError, "l4_long_ipc_send(threadid, snd_dword0,
          snd_dword1, snd_str) need three integers and one string as argument");
        return error;
    }
    _dest = (l4_threadid_t *) & threadid;
    _snd_msg.rcv_fpage.fpage = 0;
    _snd_msg.size = L4_IPC_DOPE(1,1);
    _snd_msg.send = L4_IPC_DOPE(1,1);
    _snd_msg.buf.snd_size = len;
    _snd_msg.buf.snd_str = (l4_umword_t)_snd_str;
    _timeout = L4_IPC_NEVER;

    ret = l4_ipc_send(*_dest, &_snd_msg, _snd_dword0, _snd_dword1, _timeout, &result);

    list = PyList_New((Py_ssize_t) 0);
    PyList_Append(list, Py_BuildValue("i", ret));
    Py_END_ALLOW_THREADS
    return list;
}
PyObject *call_l4_long_ipc_receive(PyObject * self, PyObject * args)
{
    [...]

    Py_BEGIN_ALLOW_THREADS
    if (!PyArg_ParseTuple(args, "i", &threadid)) {
        PyErr_SetString(PyExc_TypeError, "l4_long_ipc_receive(threadid)
          needs one integer as argument");
        return error;
    }
    _src = (l4_threadid_t *) & threadid;
    _rcv_msg.rcv_fpage.fpage = 0;
    _rcv_msg.size = L4_IPC_DOPE(1,1);
    _rcv_msg.send = L4_IPC_DOPE(1,1);
    _rcv_msg.buf.rcv_size = MAX_BUFFER_LEN ;
    _rcv_msg.buf.rcv_str = (l4_umword_t)_rcv_str;
    _timeout = L4_IPC_NEVER;
    ret = l4_ipc_receive(*_src, &_rcv_msg, &_rcv_dword0, &_rcv_dword1, _timeout,
                         &result);
    *(_rcv_str + _rcv_msg.buf.snd_size) = '\0';

    list = PyList_New((Py_ssize_t) 0);
    PyList_Append(list, Py_BuildValue("i", ret));
    PyList_Append(list, Py_BuildValue("i", _rcv_dword0));
    PyList_Append(list, Py_BuildValue("i", _rcv_dword1));
    PyList_Append(list, Py_BuildValue("s", _rcv_str));
    Py_END_ALLOW_THREADS
    return list;
}
```

Abbildung 3.9: Implementation der Sende- und Empfangsfunktionen für Long-IPC

Kapitel 4

Auswertung

Um meine Portierung zu Evaluieren, d.h. eine Aussage über Qualität und Geschwindigkeit zu machen, habe ich Unittests und ein Benchmark für Python verwendet.

Alle in den folgenden Abschnitten genannten Testergebnisse sind auf dem selben Testrechner durchgeführt worden:

- Intel Core 2 CPU T7400 2.16GHz CPU
- 2 GB RAM
- Linux Debian 4.0 x86
- Fiasco Rev. 31540 x86

4.1 Unittests

PyUnit ([PyUnit08]) ist das Standard-Unittest-Framework für Python; es ermöglicht Unittest für den Python-Interpreter, für *extension modules* und für in Python geschriebene Programme.

Die Implementation basiert auf JUnit (Test-Framework für Java) und gehört seit Python 2.1 zu Pythons Standardbibliothek. PyUnit bringt über 300 Unittests für Python mit, z.B. für eingebaute Datentypen, für Threads und für die Standardbibliothek.

Um Aussagen über die Qualität meiner Portierung machen zu können, habe ich viele Unittests für L4Python angepasst und ausgeführt. Ein Teil der Unittests sind für L4Python nicht interessant, da z.B. Datenbank-, Audio-, und Grafik-Unterstützung für Python unter L4 nicht verfügbar ist.

Aufgrund der großen Anzahl von Unittests musste ich mich auf einen Teil beschränken. Mein Wahl fiel vor allem auf Tests die die Python-Implementation testen. Weiterhin wählte ich Tests für wichtige und oft genutzte Bibliotheken aus. Insgesamt laufen 120 Unittests erfolgreich, 44 laufen nicht und 154 habe ich nicht getestet. Abbildung 4.1 zeigt alle erfolgreich gelaufenen Tests. Tests die nicht erfolgreich getestet werden konnten sind in Abbildung 4.2 aufgeführt.

Bei fast allen Tests mussten Anpassungen vorgenommen werden. Da der `simple_file_server` keine Unterverzeichnisse unterstützt, mussten die `import`-Anweisungen meist verändert werden. Ursachen für Fehlgeschlagene Tests sind zum größten Teil die Einschränkungen des Dateiservers (`simple_file_server`) von L4. Mit diesem können nur etwa 30 Dateien verfügbar gemacht werden, viele Tests benötigen aber weit aus mehr. Fehlende Bibliotheken und Funktionalitäten unter L4 sind ein weitere Grund für fehlgeschlagene Tests.

4.2 Performance von Python auf L4

Bei einer Portierung ist die Performance auf dem neuen System von besonderem Interesse. Python selbst bringt ein Benchmark – Pybench – mit.

Pybench ist ein Benchmark Suite für Python. Es besteht aus einer Sammlung von Tests, die es gemeinsam ermöglichen, verschiedene Python-Implementation bzgl. Geschwindigkeit zu bewerten. Pybench wird u.a. benutzt, um Geschwindigkeitsprobleme und finden und neue Implementation von Funktionen zu evaluieren. Pybench bewertet viele verschiedene Aspekte von Python und liefert eine detaillierte Bewertung.

Um das Ergebnis von Pybench unter L4 bewerten zu können, vergleiche ich die Ausführungszeiten mit denen unter Linux. Abbildung 4.3 zeigt die Ergebnisse des Benchmarks für L4Python und Linux-Python. Die Ausführungszeiten von Python unter L4 bzw. Linux liegen nahe beieinander – die Geschwindigkeit von L4Python unter L4 und Python unter Linux ist gleich. Das arithmetische Mittel der Abweichungen von L4Python und Linux-Python beträgt 10,25 Millisekunden. Der größte Geschwindigkeitsunterschied entsteht beim "ConcatUnicode"-Test (38ms) – der geringste (0-1ms) ist bei den Tests "CompareIntegers", "CompareInternedStrings", "DictWithFloatKeys", "SimpleDictManipulation", "SimpleIntFloatArithmetic", "UnicodeMappings" und "UnicodePredicates" zu beobachten.

Abbildung 4.4 zeigt den Geschwindigkeitsunterschied zwischen Python unter L4 und Linux. Dargestellt sind die relativen Unterschiede der Testzeiten von Python auf Linux bzw. L4. Die Testzeiten von Python auf L4 unterscheiden sich um maximal 30 Prozent von denen auf Linux. Der Mittelwert der Unterschiede beträgt zehn Prozent.

Zur Ausführung von Pybench benutzte ich Fiasco ohne Multiprozessor-Unterstützung. Um die Ergebnisse mit denen von Pybench auf Linux vergleichen zu können, habe ich den Prozessor im *Single-Core*-Modus betrieben. Im *Dual-Core*-Modus ist Pybench unter Linux fast doppelt so schnell.

Die Performance ist nicht nur von der Python-Implementation, sondern auch von der Implementation benutzter Bibliotheken abhängig. L4Python benutzt als C-Bibliothek die uClibc; Python unter Linux hingegen benutzt die GNU C-Bibliothek. Der Vergleich ist trotzdem aussagekräftig, da die genannten Bibliotheken die Standardbibliotheken unter den jeweiligen System sind und auch genutzt werden.

testname	description	testname	description
__future__	test various flavors of future statements	hashlib	tests for many hash functions.
anydbm	generic interface to all dbm clones	hexoct	test for hex/oct constants
array	array object implementation	imp	test import statement
ast	functions to transform a syntax trees	isinstance	tests for isinstance(), issubclass()
atexit	exit functions for program termination	iter	test iterators
audioop	module to detect peak values in arrays	iterlen	test Iterator Length Transparency
augassign	assigment Tests	itertools	tests for itertools module
base64	RFC 3548: Base16, Base32, Base64	list	tests for list implementation
bigaddrspace	big-memory-test support	long	test long implementation
bigmem	big-memory-test support	long_future	test long implementation
binascii	routines to represent binary data in ASCII	longexp	test long implementation
binop	tests for binary operators	marshal	write/read Python objects to/from files
bool	properties of bool promised by PEP 285	md5	testing md5 module
bufio	fileobject deliver expected results	module	test the module type
builtin	built-in functions	new	test new operator
calendar	built-in functions	opcodes	test opcodes
call	paths through the function calling	operations	test built-in operations
capi	C Extension module	operator	tests the Operator interface
class	Python classes	parser	test the parser module
cmath	math functions	pickle	serialized representations of objects
codeop	compile incomplete Python source code	profile	class for profiling python code
complex	complex Math	random	random variable generators
complex_args	complex Arguments	re	secret Labs' Regular Expression Engine
copy	generic copying operations	richcmp	tests for rich comparisons
copy_reg	helper for pickle/cPickle	set	classes to represent arbitrary sets
cpickle	serialized representations of objects	sha	testing sha module
cProfile	Python interface for the 'lsprof' profiler	slice	tests for slice objects
decorators	evaluating decorated functions	sort	test sort
defaultdict	tests for collections.defaultdict	str	tests for string implementation
descr	test descriptors and new-style classes	string	tests for string implementation
dict	tests for the dictionary	StringIO	tests for StringIO module
dl	dl module	stringprep	tests for StringPrep RFC 3454
dummy_thread	drop-in replacement for thread module	strop	tests for strop module
dummy_threading	faux threading	struct	test struct module
enumerate	tests for enumerate objects	symtable	compiler's internal symbol tables
eof	test for a few new invalid token catches	textwrap	text wrapping and filling.
errno	errno module	thread	threads
exception_variations	built-in exceptions	threading	threads- und Locks
exceptions	built-in exceptions	tokenize	tokenization help for Python programs
float	float number test	trace	testing the line trace facility
format	test string formatting operator	tuple	test tuple implementation
fpformat	floating point formatting functions	types	built-in types
frozen	test the frozen module	unary	test unary ops (+, -, ~)
funcattrs	attributes on functions	univnewlines	universal newline
functools	tools for functions, callable objects	userdict	tests for UserDict
future	test future statements	userlist	tests for UserList
gc	Reference Cycle Garbage Collection	userstring	test for userString module
getargs	test the getargs.c implementation	uuid	UUID objects according to RFC 4122.
getargs2	test the getargs.c implementation	warnings	Python warnings subsystem
global	warnings for global statements	weakref	weak reference support for Python
grammar	tests grammar	with	tests for the with in PEP 343
hash	test iff a==b then hash(a)==hash(b)	xpickle	tests for xpickle module

tests without description

codeccallbacks, coding, coercion, compare, compile, contains, contextlib, extcall
index, peepholer, pow, profilehooks, scope, softspace, structmembers, xrange

Abbildung 4.1: Erfolgreich gelaufene Unittests

testname	description	reason for failure
asynchat	asynchrone Sockets	Flips & DDE needed
binhex	binhex compression/decompression	bad file deskriptor
bz2	bzip2	no bz2 lib
cfgparser	Configuration file parser	unsupported file operations
cmd_line		unimplemented functions: pipe,sysconf
dircache	Read and cache directory listings	tmpfs does not support creating directories
fcntl	fcntl() locking calls	fcntl(): F_GETLK, F_SETLK, F_SETLKW not supported
filecmp	Utilities for comparing files and directories	tmpfs does not support creating directories
gettext	Internationalization and localization support	unsupported file operations
glob	Filename globbing utility.	tmpfs does not support creating directories
hotshot	High-perfomance logging profiler	unimplemented function: getrusage
import	test import statement	Bad magic number in .//tmp/test.pyc
ioctl	tests for ioctl	unimplemented function: termios_tcflush
math	Test math module	Sqrt(-1) don't raise ValueError
mmap	map a view of a file into memory	unimplemented function: ftruncate
pkg	Tests for import functionallity	unsupported file operations
pkgimport	Tests for import functionallity	unsupported file operations
poll	Test case for the os.poll() function	select.poll not defined
posix	Test posix functions	unsupported file operations
pty	Pseudo terminal utilities.	unimplemented function: termios_tcflush
repr	tests for repr module	unsupported file operations
resource	tests for resource module	unimplemented function: setrlimit
runpy	locating, running Python code	unsupported file operations
select	Unix select(2) call.	unimplemented function: popen
shlex	A lexical analyzer class	encoding problem
signal	Signal module	unimplemented function: fork
site	tests for site module	unsupported file operations
strftime	Sanity checker for time.strftime	Unimplemented function: gmtime
strptime	Test for strptime	unimplemented function: gmtime
structseq	tests for structseq	unimplemented function: gmtime
sys	Tests for the system module	unimplemented functions: pipe,sysconf
tempfile	Temporary files.	unsupported file operations
threaded_import	tests for thread implementation	unsupported file operations
threadedtempfile	tests for thread implementation	unsupported file operations
threadsignals	Testing that threads honor signal semantics	unimplemented function: getpid
time	Tests the time module	unimplemented function: gmtime
traceback	extract information about stack traces	unsupported file operations
wait3	checks for correct wait3() behavior	unimplemented function: fork

Abbildung 4.2: Nicht erfolgreich gelaufene Unittests

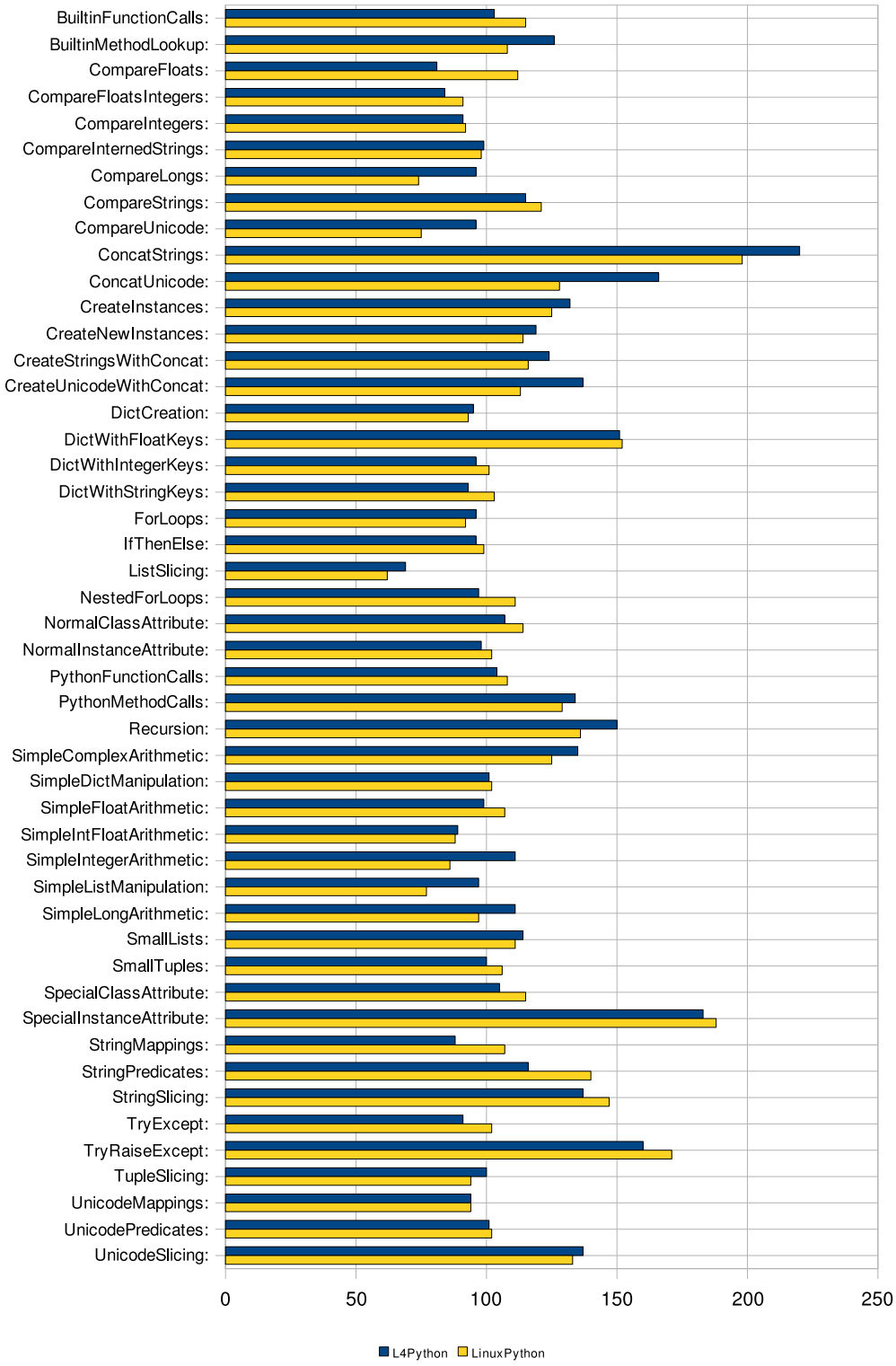

Abbildung 4.3: absolute Testzeiten (in ms) von Pybench auf Linux bzw. L4

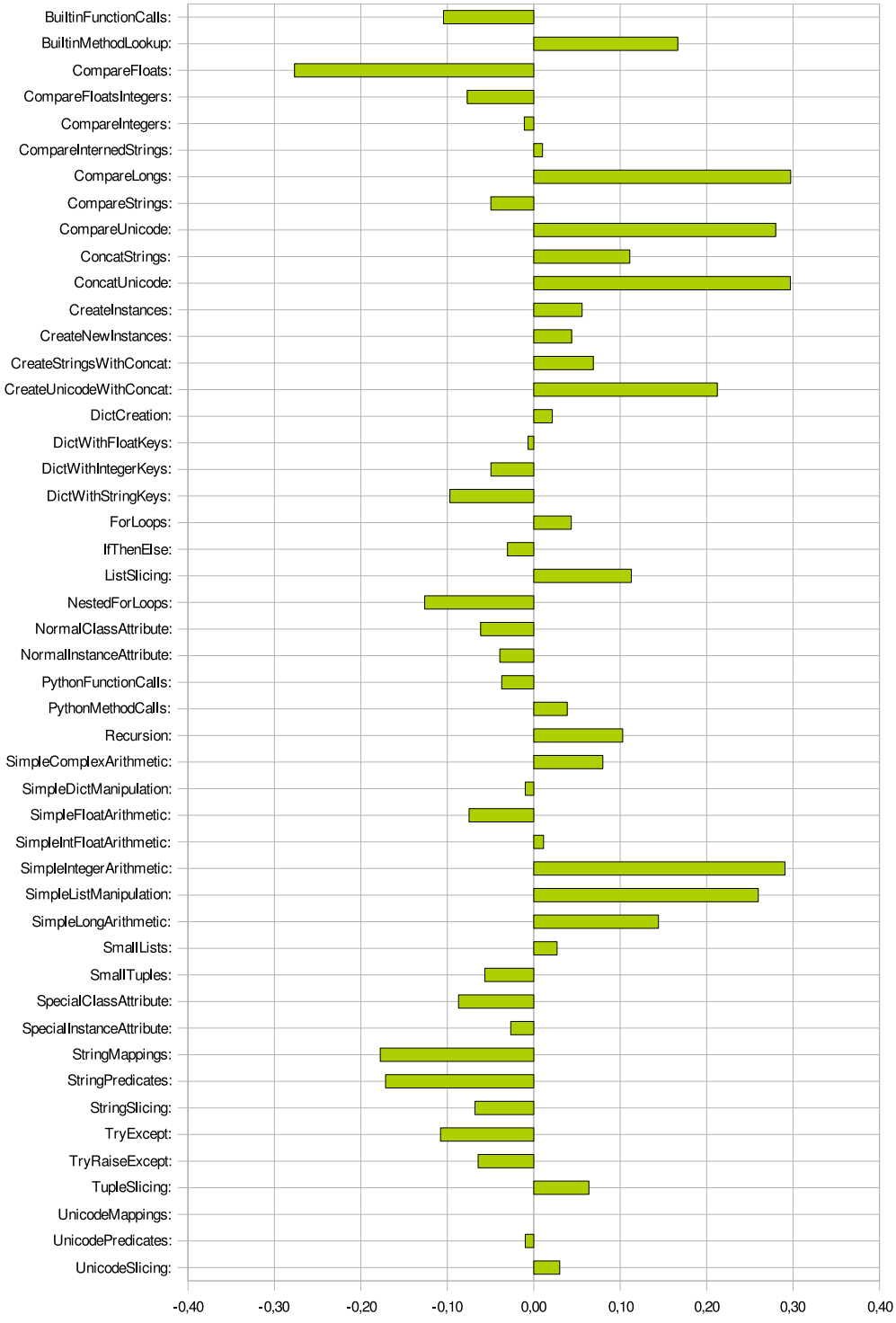

Abbildung 4.4: relative Unterschiede (in %) der Testzeiten von Python auf Linux bzw. L4

Kapitel 5

Ausblick

Meine Arbeit zeigt Möglichkeiten, L4 Bibliotheken in Python einzubinden. Denkbar wäre eine DICE-Schnittstelle für Python; damit könnten *extension modules* direkt aus den IDL-Definitionen der L4 Server generiert werden. Wie in meiner Arbeit wären bei der Implementation einer solchen Schnittstelle die Typumwandlungen von C/C++ in Python Typen und umgekehrt ein zentrales Problem.

Python verfügt über Module zur Nutzung grafischer Oberflächen, zum Beispiel Qt, Tk, GTK. Es wäre denkbar, eine Anbindung von Python an das bereits portierte Qt3[Wei05] zu implementieren. Möglich wären auch Portierungen anderer grafischer Oberflächen auf L4, um diese mit Python zu nutzen.

Ein paar *extension modules* von Python ließen sich nicht auf L4 portieren, da zum Beispiel benötigte Bibliotheken fehlten. Um diese zu portieren müssten benötigte Bibliotheken implementiert bzw. portiert werden.

Um L4Python zum Starten von Programmen zu verwenden, müsste eine Anbindung an den Loader geschaffen werden. Weiterhin ist dazu ein Dateiserver nötig, der keine Beschränkungen bzgl. Anzahl der bereitstellbaren Dateien hat (Vergleich `simple_file_servers` 1.2).

Um die Leistungsfähigkeit von L4Python auf L4 zu demonstrieren, könnte ein Server vollständig in Python implementiert werden. Der Server bietet einen Dienst an, welchen Clients in Anspruch nehmen können. Da die Kommunikation auf L4 per IPC erfolgt, müsste entweder eine DICE-Schnittstelle für Python implementiert oder meine IPC-Implementation (siehe 3.3.4) vervollständigt werden.

Python 3000 (Python 3.0) ist eine, in Entwicklung befindliche, Überarbeitung der Programmiersprache Python. Um die neuen Spracheigenschaften nutzen zu können, müsste Python 3.0 auf L4 portiert werden.

Auf *Dual-Core*-Prozessoren ist Python schneller als auf *Single-Core*-Prozessoren. Denkbar wäre eine Beobachtung der Geschwindigkeit von Python auf Fiasco mit Multiprozessor-Unterstützung. Dadurch könnte u.a. die Multiprozessor-Performance von Fiasco bewertet werden.

Kapitel 6

Zusammenfassung

Skriptsprachen ermöglichen einfache und schnelle Implementationen – da der Quelltext nicht übersetzt werden muss verkürzt sich die Entwicklungszeit. Durch Konzepte wie die Erweiterbarkeit sind Skriptsprachen genauso mächtig wie richtige Programmiersprachen (z.B. C, C++).

Shells bzw. Skriptsprachen werden zum Steuern und Bedienen von Betriebssystemen verwendet(z.B. Bash unter GNU/Linux). Weiterhin werden sie oft als Brücke zwischen unterschiedlichen Programmen, die nicht zur Kommunikation untereinander ausgelegt sind, verwendet.

Python ist ein Vertreter der Skriptsprachen. Ursprünglich zur Steuerung und Administration des Betriebsystems Amoeba gedacht, wird es heute zur Implementation komplexer Systeme (Zope) bzw. zur schnellen Entwicklung von Skripten benutzt.

L4Python ist eine Portierung von Python 2.5.1. auf L4. Zur effektiven Nutzung kann L4Python mit L4 Bibliotheken erweitert werden, wodurch mit anderen L4 Komponenten kommuniziert werden kann. Weiterhin können Pythonskripte, Systemaufrufe tätigen. Im Kontext von L4 sind das vor allem IPC-Aufrufe.

Das Erweitern von Python mit L4 Bibliotheken kann per Hand oder mit Hilfe eines Werzeugs (SIP, SWIG) geschehen, in beiden Fällen wird ein *extension module* erstellt. Die erste Möglichkeit, Python per Hand zu erweitern, bietet maximale Flexibilität, erfordert aber den meisten Aufwand. Die zweite Möglichkeit, Python mit Hilfe eines Werkzeugs zu erweitern, ermöglicht eine fast vollständige automatische Generierung des *extension module*.

Insgesamt bietet L4Python Möglichkeiten zur Automatisierung von Abläufen, zur Durchführung von Administrationsaufgaben bzw. zum Test von L4 Komponenten. Es stellt eine beliebig erweiterbare Schnittstelle für den Benutzer zum System dar.

Literaturverzeichnis

[Bash08] Bash,
http://www.gnu.org/software/bash, 2008

[Ruby08] Ruby,
http://www.ruby-lang.org, 2008

[Lua08] Lua,
http://www.lua.org, 2008

[IFW96] R. Ierusalimschy, L. H. de Figueiredo, W. Celes,
Lua – an extensible extension language,
Software: Practice & Experience 26 #6 (1996) 635-652. [doi]
http://www.lua.org/spe.html

[Perl08] Perl,
http://www.perl.org, 2008

[Python08] Python,
http://www.python.org, 2008

[LangStudy08] Scriptometer: measuring the ease of SOP (Script-Oriented Programming) of programming languages,
http://merd.sourceforge.net/pixel/language-study/scripting-language/, 2008

[Ouster98] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century, 1998 IEEE Computer magazine
http://home.pacbell.net/ouster/scripting.html, 2008

[uClibc08] uClibc,
http://www.uclibc.org/, 2008

[Posix08] POSIX,
http://standards.ieee.org/regauth/posix/, 2008

[Yapps08] Yapps (Yet Another Python Parser System)
http://theory.stanford.edu/~amitp/yapps, 2008

[L408] Operating Systems Group Technische Universität Dresden,
The l4 environment,
www.tudos.org/l4env.

[Wei05] Carsten Weinhold. Portierung von Qt auf DROPS. Großer Beleg, Technische Universität Dresden, Lehrstuhl für Betriebsysteme, 2005
http://os.inf.tu-dresden.de/papers_ps/weinhold-beleg.pdf

[SIP08] SIP,
http://www.riverbankcomputing.co.uk/sip/index.php, 2008

[SWIG08] SWIG,
http://www.swig.org/, 2008

[PyCXX08] PyCXX Python-C++ Connection,
http://sourceforge.net/projects/cxx/, 2008

[Boost08] Boost C++ Libraries,
http://www.boost.org/, 2008

[Pyrex08] Pyrex – a Language for Writing Python Extension Modules,
http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/, 2008

[PyUnit08] PyUnit – the standard unit testing framework for Python,
http://pyunit.sourceforge.net/, 2008