SYSTEM AND LIBRARY CALLS EVERY PROGRAMMER NEEDS TO KNOW

# LINUX SYSTEM PROGRAMMING

O'REILLY®

ROBERT LOVE

# LINUX
## System Programming

## Other Linux resources from O'Reilly

**Related titles**

- Building Embedded Linux Systems
- Designing Embedded Hardware
- Linux Device Drivers
- Linux Kernel in a Nutshell
- Programming Embedded Systems
- Running Linux
- Understanding Linux Network Internals
- Understanding the Linux Kernel

**Linux Books Resource Center**

*linux.oreilly.com* is a complete catalog of O'Reilly's books on Linux and Unix and related technologies, including sample chapters and code examples.


*ONLamp.com* is the premier site for the open source web platform: Linux, Apache, MySQL and either Perl, Python, or PHP.

**Conferences**

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.


Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

# LINUX
## System Programming

*Robert Love*

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

**Linux System Programming**
by Robert Love



Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editor:** Andy Oram
**Production Editor:** Sumita Mukherji
**Copyeditor:** Rachel Head
**Proofreader:** Sumita Mukherji
**Indexer:** John Bickelhaupt
**Cover Designer:** Karen Montgomery
**Interior Designer:** David Futato
**Illustrator:** Jessamyn Read

**Printing History:**

September 2007: First Edition.




This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-00958-5
ISBN-13: 978-0-596-00958-8
[M]

# Table of Contents

# Foreword

There is an old line that Linux kernel developers like to throw out when they are feeling grumpy: "User space is just a test load for the kernel."

By muttering this line, the kernel developers aim to wash their hands of all responsibility for any failure to run user-space code as well as possible. As far as they're concerned, user-space developers should just go away and fix their own code, as any problems are definitely not the kernel's fault.

To prove that it usually is not the kernel that is at fault, one leading Linux kernel developer has been giving a "Why User Space Sucks" talk to packed conference rooms for more than three years now, pointing out real examples of horrible user-space code that everyone relies on every day. Other kernel developers have created tools that show how badly user-space programs are abusing the hardware and draining the batteries of unsuspecting laptops.

But while user-space code might be just a "test load" for kernel developers to scoff at, it turns out that all of these kernel developers also depend on that user-space code every day. If it weren't present, all the kernel would be good for would be to print out alternating ABABAB patterns on the screen.

Right now, Linux is the most flexible and powerful operating system that has ever been created, running everything from the tiniest cell phones and embedded devices to more than 70 percent of the world's top 500 supercomputers. No other operating system has ever been able to scale so well and meet the challenges of all of these different hardware types and environments.

And along with the kernel, code running in user space on Linux can also operate on all of those platforms, providing the world with real applications and utilities people rely on.

In this book, Robert Love has taken on the unenviable task of teaching the reader about almost every system call on a Linux system. In so doing, he has produced a tome that will allow you to fully understand how the Linux kernel works from a user-space perspective, and also how to harness the power of this system.

The information in this book will show you how to create code that will run on all of the different Linux distributions and hardware types. It will allow you to understand how Linux works and how to take advantage of its flexibility.

In the end, this book teaches you how to write code that doesn't suck, which is the best thing of all.

—Greg Kroah-Hartman

# Preface

This book is about system programming—specifically, system programming on Linux. *System programming* is the practice of writing *system software*, which is code that lives at a low level, talking directly to the kernel and core system libraries. Put another way, the topic of the book is Linux system calls and other low-level functions, such as those defined by the C library.

While many books cover system programming for Unix systems, few tackle the subject with a focus solely on Linux, and fewer still (if any) address the very latest Linux releases and advanced Linux-only interfaces. Moreover, this book benefits from a special touch: I have written a lot of code for Linux, both for the kernel and for system software built thereon. In fact, I have implemented some of the system calls and other features covered in this book. Consequently, this book carries a lot of insider knowledge, covering not just how the system interfaces *should* work, but how they *actually* work, and how you (the programmer) can use them most efficiently. This book, therefore, combines in a single work a tutorial on Linux system programming, a reference manual covering the Linux system calls, and an insider's guide to writing smarter, faster code. The text is fun and accessible, and regardless of whether you code at the system level on a daily basis, this book will teach you tricks that will enable you to write better code.

## Audience and Assumptions

The following pages assume that the reader is familiar with C programming and the Linux programming environment—not necessarily well-versed in the subjects, but at least acquainted with them. If you have not yet read any books on the C programming language, such as the classic Brian W. Kernighan and Dennis M. Ritchie work *The C Programming Language* (Prentice Hall; the book is familiarly known as K&R), I highly recommend you check one out. If you are not comfortable with a Unix text editor—Emacs and *vim* being the most common and highly regarded—start playing

with one. You'll also want to be familiar with the basics of using *gcc*, *gdb*, *make*, and so on. Plenty of other books on tools and practices for Linux programming are out there; the bibliography at the end of this book lists several useful references.

I've made few assumptions about the reader's knowledge of Unix or Linux system programming. This book will start from the ground up, beginning with the basics, and winding its way up to the most advanced interfaces and optimization tricks. Readers of all levels, I hope, will find this work worthwhile and learn something new. In the course of writing the book, I certainly did.

Nor do I make assumptions about the persuasion or motivation of the reader. Engineers wishing to program (better) at a low level are obviously targeted, but higher-level programmers looking for a stronger standing on the foundations on which they rest will also find a lot to interest them. Simply curious hackers are also welcome, for this book should satiate their hunger, too. Whatever readers want and need, this book should cast a net wide enough—as least as far as Linux system programming is concerned—to satisfy them.

Regardless of your motives, above all else, *have fun*.

## Contents of This Book

This book is broken into 10 chapters, an appendix, and a bibliography.

Chapter 1, *Introduction and Essential Concepts*
: This chapter serves as an introduction, providing an overview of Linux, system programming, the kernel, the C library, and the C compiler. Even advanced users should visit this chapter—trust me.

Chapter 2, *File I/O*
: This chapter introduces files, the most important abstraction in the Unix environment, and file I/O, the basis of the Linux programming mode. This chapter covers reading from and writing to files, along with other basic file I/O operations. The chapter culminates with a discussion on how the Linux kernel implements and manages files.

Chapter 3, *Buffered I/O*
: This chapter discusses an issue with the basic file I/O interfaces—buffer size management—and introduces buffered I/O in general, and standard I/O in particular, as solutions.

Chapter 4, *Advanced File I/O*
: This chapter completes the I/O troika with a treatment on advanced I/O interfaces, memory mappings, and optimization techniques. The chapter is capped with a discussion on avoiding seeks, and the role of the Linux kernel's I/O scheduler.

Chapter 5, *Process Management*
: This chapter introduces Unix's second most important abstraction, the *process*, and the family of system calls for basic process management, including the venerable *fork*.

Chapter 6, *Advanced Process Management*
: This chapter continues the treatment with a discussion of advanced process management, including real-time processes.

Chapter 7, *File and Directory Management*
: This chapter discusses creating, moving, copying, deleting, and otherwise managing files and directories.

Chapter 8, *Memory Management*
: This chapter covers memory management. It begins by introducing Unix concepts of memory, such as the process address space and the page, and continues with a discussion of the interfaces for obtaining memory from and returning memory to the kernel. The chapter concludes with a treatment on advanced memory-related interfaces.

Chapter 9, *Signals*
: This chapter covers signals. It begins with a discussion of signals and their role on a Unix system. It then covers signal interfaces, starting with the basic, and concluding with the advanced.

Chapter 10, *Time*
: This chapter discusses time, sleeping, and clock management. It covers the basic interfaces up through POSIX clocks and high-resolution timers.

Appendix, *GCC Extensions to the C Language*
: The Appendix reviews many of the optimizations provided by *gcc* and GNU C, such as attributes for marking a function constant, pure, and inline.

The book concludes with a bibliography of recommended reading, listing both useful supplements to this work, and books that address prerequisite topics not covered herein.

## Versions Covered in This Book

The Linux system interface is definable as the application binary interface and application programming interface provided by the triplet of the Linux kernel (the heart of the operating system), the GNU C library (*glibc*), and the GNU C Compiler (*gcc*—now formally called the GNU Compiler Collection, but we are concerned only with C). This book covers the system interface defined by Linux kernel version 2.6.22, *glibc* version 2.5, and *gcc* version 4.2. Interfaces in this book should be backward compatible with older versions (excluding new interfaces), and forward compatible to newer versions.

If any evolving operating system is a moving target, Linux is a rabid cheetah. Progress is measured in days, not years, and frequent releases of the kernel and other components constantly morph the playing field. No book can hope to capture such a dynamic beast in a timeless fashion.

Nonetheless, the programming environment defined by system programming is *set in stone*. Kernel developers go to great pains not to break system calls, the *glibc* developers highly value forward *and* backward compatibility, and the Linux toolchain generates compatible code across versions (particularly for the C language). Consequently, while Linux may be constantly on the go, Linux system programming remains stable, and a book based on a snapshot of the system, especially at this point in Linux's development, has immense staying power. What I am trying to say is simple: don't worry about system interfaces changing, and *buy this book*!

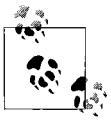
## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Used for emphasis, new terms, URLs, foreign phrases, Unix commands and utilities, filenames, directory names, and pathnames.

`Constant width`
: Indicates header files, variables, attributes, functions, types, parameters, objects, macros, and other programming constructs.

*`Constant width italic`*
: Indicates text (for example, a pathname component) to be replaced with a user-supplied value.



This icon signifies a tip, suggestion, or general note.

Most of the code in this book is in the form of brief, but usable, code snippets. They look like this:

```
while (1) {
        int ret;

        ret = fork ( );
        if (ret == -1)
                perror ("fork");
}
```

Great pains have been taken to provide code snippets that are concise but usable. No special header files, full of crazy macros and illegible shortcuts, are required. Instead of building a few gigantic programs, this book is filled with many simple examples.

As the examples are descriptive and fully usable, yet small and clear, I hope they will provide a useful tutorial on the first read, and remain a good reference on subsequent passes.

Nearly all of the examples in this book are self-contained. This means you can easily copy them into your text editor, and put them to actual use. Unless otherwise mentioned, all of the code snippets should build without any special compiler flags. (In a few cases, you need to link with a special library.) I recommend the following command to compile a source file:

```
$ gcc -Wall -Wextra -O2 -g -o snippet snippet.c
```

This compiles the source file *snippet.c* into the executable binary *snippet*, enabling many warning checks, significant but sane optimizations, and debugging. The code in this book should compile using this command without errors or warnings—although of course, you might have to build a skeleton program around the snippet first.

When a section introduces a new function, it is in the usual Unix manpage format with a special emphasized font, which looks like this:

```
#include <fcntl.h>

int posix_fadvise (int fd, off_t pos, off_t len, int advice);
```

The required headers, and any needed definitions, are at the top, followed by a full prototype of the call.

## Safari® Books Online


When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

## Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you are reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting

example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Linux System Programming* by Robert Love. Copyright 2007 O'Reilly Media, Inc., 978-0-596-00958-8."

If you believe that your use of code examples falls outside of fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at this address:

*http://www.oreilly.com/catalog/9780596009588/*

To comment or ask technical questions about this book, you can send an email to the following address:

*bookquestions@oreilly.com*

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at this address:

*http://www.oreilly.com*

## Acknowledgments

Many hearts and minds contributed to the completion of this manuscript. While no list would be complete, it is my sincere pleasure to acknowledge the assistance and friendship of individuals who provided encouragement, knowledge, and support along the way.

Andy Oram is a phenomenal editor and human being. This effort would have been impossible without his hard work. A rare breed, Andy couples deep technical knowledge with a poetic command of the English language.

Brian Jepson served brilliantly as editor for a period, and his sterling efforts continue to reverberate throughout this work as well.

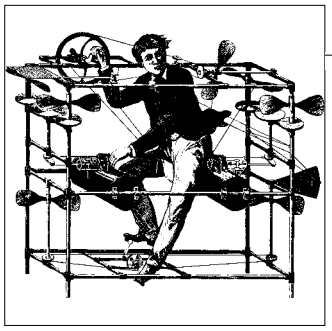
This book was blessed with phenomenal technical reviewers, true masters of their craft, without whom this work would pale in comparison to the final product you now read. The technical reviewers were Robert Day, Jim Lieb, Chris Rivera, Joey Shaw, and Alain Williams. Despite their toils, any errors remain my own.

Rachel Head performed flawlessly as copyeditor. In her aftermath, red ink decorated my written word—readers will certainly appreciate her corrections.

For numerous reasons, thanks and respect to Paul Amici, Mikey Babbitt, Keith Barbag, Jacob Berkman, Dave Camp, Chris DiBona, Larry Ewing, Nat Friedman, Albert Gator, Dustin Hall, Joyce Hawkins, Miguel de Icaza, Jimmy Krehl, Greg Kroah-Hartman, Doris Love, Jonathan Love, Linda Love, Tim O'Reilly, Aaron Matthews, John McCain, Randy O'Dowd, Salvatore Ribaudo and family, Chris Rivera, Joey Shaw, Sarah Stewart, Peter Teichman, Linus Torvalds, Jon Trowbridge, Jeremy VanDoren and family, Luis Villa, Steve Weisberg and family, and Helen Whisnant.

Final thanks to my parents, Bob and Elaine.

—Robert Love
*Boston*



CHAPTER 1

# Introduction and Essential Concepts

This book is about *system programming*, which is the art of writing *system software*. System software lives at a low level, interfacing directly with the kernel and core system libraries. System software includes your shell and your text editor, your compiler and your debugger, your core utilities and system daemons. These components are entirely system software, based on the kernel and the C library. Much other software (such as high-level GUI applications) lives mostly in the higher levels, delving into the low level only on occasion, if at all. Some programmers spend all day every day writing system software; others spend only part of their time on this task. There is no programmer, however, who does not benefit from some understanding of system programming. Whether it is the programmer's *raison d'être*, or merely a foundation for higher-level concepts, system programming is at the heart of all software that we write.

In particular, this book is about system programming on *Linux*. Linux is a modern Unix-like system, written from scratch by Linus Torvalds, and a loose-knit community of hackers around the globe. Although Linux shares the goals and ideology of Unix, Linux is not Unix. Instead, Linux follows its own course, diverging where desired, and converging only where practical. Generally, the core of Linux system programming is the same as on any other Unix system. Beyond the basics, however, Linux does well to differentiate itself—in comparison with traditional Unix systems, Linux is rife with additional system calls, different behavior, and new features.

## System Programming

Traditionally speaking, all Unix programming is system-level programming. Historically, Unix systems did not include many higher-level abstractions. Even programming in a development environment such as the X Window System exposed in full view the core Unix system API. Consequently, it can be said that this book is a book on Linux

programming in general. But note that this book does not cover the Linux programming *environment*—there is no tutorial on *make* in these pages. What is covered is the system programming API exposed on a modern Linux machine.

System programming is most commonly contrasted with application programming. System-level and application-level programming differ in some aspects, but not in others. System programming is distinct in that system programmers must have a strong awareness of the hardware and operating system on which they are working. Of course, there are also differences between the libraries used and calls made. Depending on the "level" of the stack at which an application is written, the two may not actually be very interchangeable, but, generally speaking, moving from application programming to system programming (or vice versa) is not hard. Even when the application lives very high up the stack, far from the lowest levels of the system, knowledge of system programming is important. And the same good practices are employed in all forms of programming.

The last several years have witnessed a trend in application programming away from system-level programming and toward very high-level development, either through web software (such as JavaScript or PHP), or through managed code (such as C# or Java). This development, however, does not foretell the death of system programming. Indeed, someone still has to write the JavaScript interpreter and the C# runtime, which is itself system programming. Furthermore, the developers writing PHP or Java can still benefit from knowledge of system programming, as an understanding of the core internals allows for better code no matter where in the stack the code is written.

Despite this trend in application programming, the majority of Unix and Linux code is still written at the system level. Much of it is C, and subsists primarily on interfaces provided by the C library and the kernel. This is traditional system programming—Apache, *bash*, *cp*, Emacs, *init*, *gcc*, *gdb*, *glibc*, *ls*, *mv*, *vim*, and X. These applications are not going away anytime soon.

The umbrella of system programming often includes kernel development, or at least device driver writing. But this book, like most texts on system programming, is unconcerned with kernel development. Instead, it focuses on user-space system-level programming; that is, everything above the kernel (although knowledge of kernel internals is a useful adjunct to this text). Likewise, network programming—sockets and such—is not covered in this book. Device driver writing and network programming are large, expansive topics, best tackled in books dedicated to the subject.

What is the system-level interface, and how do I write system-level applications in Linux? What exactly do the kernel and the C library provide? How do I write optimal code, and what tricks does Linux provide? What neat system calls are provided in Linux compared to other Unix variants? How does it all work? Those questions are at the center of this book.

There are three cornerstones to system programming in Linux: system calls, the C library, and the C compiler. Each deserves an introduction.

## System Calls

System programming starts with *system calls*. System calls (often shorted to *syscalls*) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system. System calls range from the familiar, such as `read()` and `write()`, to the exotic, such as `get_thread_area()` and `set_tid_address()`.

Linux implements far fewer system calls than most other operating system kernels. For example, a count of the i386 architecture's system calls comes in at around 300, compared with the allegedly thousands of system calls on Microsoft Windows. In the Linux kernel, each machine architecture (such as Alpha, i386, or PowerPC) implements its own list of available system calls. Consequently, the system calls available on one architecture may differ from those available on another. Nonetheless, a very large subset of system calls—more than 90 percent—is implemented by all architectures. It is this shared subset, these common interfaces, that I cover in this book.

### Invoking system calls

It is not possible to directly link user-space applications with kernel space. For reasons of security and reliability, user-space applications must not be allowed to directly execute kernel code or manipulate kernel data. Instead, the kernel must provide a mechanism by which a user-space application can "signal" the kernel that it wishes to invoke a system call. The application can then *trap* into the kernel through this well-defined mechanism, and execute only code that the kernel allows it to execute. The exact mechanism varies from architecture to architecture. On i386, for example, a user-space application executes a software interrupt instruction, `int`, with a value of `0x80`. This instruction causes a switch into kernel space, the protected realm of the kernel, where the kernel executes a software interrupt handler—and what is the handler for interrupt `0x80`? None other than the system call handler!

The application tells the kernel which system call to execute and with what parameters via *machine registers*. System calls are denoted by number, starting at 0. On the i386 architecture, to request system call 5 (which happens to be `open()`), the user-space application stuffs `5` in register `eax` before issuing the `int` instruction.

Parameter passing is handled in a similar manner. On i386, for example, a register is used for each possible parameter—registers `ebx`, `ecx`, `edx`, `esi`, and `edi` contain, in order, the first five parameters. In the rare event of a system call with more than five parameters, a single register is used to point to a buffer in user space where all of the parameters are kept. Of course, most system calls have only a couple of parameters.

Other architectures handle system call invocation differently, although the spirit is the same. As a system programmer, you usually do not need any knowledge of how the kernel handles system call invocation. That knowledge is encoded into the standard calling conventions for the architecture, and handled automatically by the compiler and the C library.

## The C Library

The C library (*libc*) is at the heart of Unix applications. Even when you're programming in another language, the C library is most likely in play, wrapped by the higher-level libraries, providing core services, and facilitating system call invocation. On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*, and pronounced *gee-lib-see* or, less commonly, *glib-see*.

The GNU C library provides more than its name suggests. In addition to implementing the standard C library, *glibc* provides wrappers for system calls, threading support, and basic application facilities.

## The C Compiler

In Linux, the standard C compiler is provided by the *GNU Compiler Collection* (*gcc*). Originally, *gcc* was GNU's version of *cc*, the C Compiler. Thus, *gcc* stood for *GNU C Compiler*. Over time, support was added for more and more languages. Consequently, nowadays *gcc* is used as the generic name for the family of GNU compilers. However, *gcc* is also the binary used to invoke the C compiler. In this book, when I talk of *gcc*, I typically mean the program *gcc*, unless context suggests otherwise.

The compiler used in a Unix system—Linux included—is highly relevant to system programming, as the compiler helps implement the C standard (see "C Language Standards") and the system ABI (see "APIs and ABIs"), both later in this chapter.

# APIs and ABIs

Programmers are naturally interested in ensuring their programs run on all of the systems that they have promised to support, now and in the future. They want to feel secure that programs they write on their Linux distributions will run on other Linux distributions, as well as on other supported Linux architectures and newer (or earlier) Linux versions.

At the system level, there are two separate sets of definitions and descriptions that impact portability. One is the *application programming interface* (API), and the other is the *application binary interface* (ABI). Both define and describe the interfaces between different pieces of computer software.

## APIs

An API defines the interfaces by which one piece of software communicates with another at the source level. It provides abstraction by providing a standard set of interfaces—usually functions—that one piece of software (typically, although not

necessarily, a higher-level piece) can invoke from another piece of software (usually a lower-level piece). For example, an API might abstract the concept of drawing text on the screen through a family of functions that provide everything needed to draw the text. The API merely defines the interface; the piece of software that actually provides the API is known as the *implementation* of the API.

It is common to call an API a "contract." This is not correct, at least in the legal sense of the term, as an API is not a two-way agreement. The API user (generally, the higher-level software) has zero input into the API and its implementation. It may use the API as-is, or not use it at all: take it or leave it! The API acts only to ensure that if both pieces of software follow the API, they are *source compatible*; that is, that the user of the API will successfully compile against the implementation of the API.

A real-world example is the API defined by the C standard and implemented by the standard C library. This API defines a family of basic and essential functions, such as string-manipulation routines.

Throughout this book, we will rely on the existence of various APIs, such as the standard I/O library discussed in Chapter 3. The most important APIs in Linux system programming are discussed in the section "Standards" later in this chapter.

## ABIs

Whereas an API defines a source interface, an ABI defines the low-level binary interface between two or more pieces of software on a particular architecture. It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries. An ABI ensures *binary compatibility*, guaranteeing that a piece of object code will function on any system with the same ABI, without requiring recompilation.

ABIs are concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object format. The calling convention, for example, defines how functions are invoked, how arguments are passed to functions, which registers are preserved and which are mangled, and how the caller retrieves the return value.

Although several attempts have been made at defining a single ABI for a given architecture across multiple operating systems (particularly for i386 on Unix systems), the efforts have not met with much success. Instead, operating systems—Linux included—tend to define their own ABIs however they see fit. The ABI is intimately tied to the architecture; the vast majority of an ABI speaks of machine-specific concepts, such as particular registers or assembly instructions. Thus, each machine architecture has its own ABI on Linux. In fact, we tend to call a particular ABI by its machine name, such as *alpha*, or *x86-64*.