CHAPTER 1

Introduction and Essential Concepts

This book is about *system programming*, which is the practice of writing *system software*. System software lives at a low level, interfacing directly with the kernel and core system libraries. Your shell and your text editor, your compiler and your debugger, your core utilities and system daemons are all system software. But so are the network server, the web server, and the database. These components are entirely system software, primarily if not exclusively interfacing with the kernel and the C library. Other software (such as high-level GUI applications) lives at a higher level, delving into the low level only on occasion. 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 an 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 programmers around the globe. Although Linux shares the goals and philosophy of Unix, Linux is not Unix. Instead, Linux follows its own course, diverging where desired and converging only where practical. The core of Linux system programming is the same as on any other Unix system. Beyond the basics, however, Linux differentiates itself—in comparison with traditional Unix systems, Linux supports additional system calls, behaves distinctly, and offers new features.

System Programming

Traditionally, all Unix programming was system-level programming. Unix systems historically 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 —for example, there is no tutorial on *make* in these pages. What is covered is the system programming API exposed on a modern Linux machine.

We can compare and contrast system programming with application programming, which differ in some important aspects but are quite similar in others. System programming's hallmark is that the system programmer must have an acute awareness of the hardware and the operating system on which they work. Where system programs interface primarily with the kernel and system libraries, application programs also interface with high-level libraries. These libraries *abstract* away the details of the hardware and operating system. Such abstraction has several goals: portability with different systems, compatibility with different versions of those systems, and the construction of higher-level toolkits that are easier to use, more powerful, or both. How much of a given application uses system versus high-level libraries depends on the level of the stack at which the application was written. Some applications are written exclusively to higher-level abstractions. But even these applications, far from the lowest levels of the system, benefit from a programmer with knowledge of system programming. The same good practices and understanding of the underlying system inform and benefit all forms of programming.

Why Learn System Programming

The preceding decade has 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 through managed code (such as Java). This development, however, does not foretell the death of system programming. Indeed, someone still has to write the JavaScript interpreter and the Java VM, which are themselves system programming. Furthermore, the developer writing Python or Ruby or Scala can still benefit from knowledge of system programming, as an understanding of the soul of the machine 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 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). Device driver writing is a large, expansive topic, 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 interesting 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.

Cornerstones of System Programming

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

System Calls

System programming starts and ends with *system calls*. System calls (often shortened 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 x86-64 architecture's system calls comes in at around 300, compared with the suspected thousands of system calls on Microsoft Windows. In the Linux kernel, each machine architecture (such as Alpha, x86-64, or PowerPC) can augment the standard system calls with its own. 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 we 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" on page 8) and the system ABI (see "APIs and ABIs" on page 5).

C++

This chapter focuses on C as the lingua franca of system programming, but C++ plays a significant role.

To date, C++ has taken a backseat to C in system programming. Historically, Linux developers favored C over C++: core libraries, daemons, utilities, and of course the Linux kernel are all written in C. Where the ascendancy of C++ as a "better C" is all but universal in most non-Linux environments, in Linux C++ plays second fiddle to C.

Nonetheless, in much of this book, you can replace "C" with "C++" without issue. Indeed, C++ is an excellent alternative to C, suitable for any system programming task: C ++ code can link to C code, invoke Linux system calls, and utilize *glibc*.

C++ programming adds two more cornerstones to the system programming foundation: the standard C++ library and the GNU C++ compiler. The *standard C++ library* implements C++ system interfaces and the ISO C++11 standard. It is provided by the *libstdc++* library (sometimes written *libstdcxx*). The *GNU C++ compiler* is the standard compiler for C++ code on Linux systems. It is provided by the *g++* binary.

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 of an API is the interfaces defined by the C standard and implemented by the standard C library. This API defines a family of basic and essential functions, such as memory management and 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" on page 7.

ABIs

Whereas an API defines a source interface, an ABI defines the 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. Whereas an API ensures source compatibility, 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*. Thus, the ABI is a function of both the operating system (say, Linux) and the architecture (say, x86-64).

System programmers ought to be aware of the ABI but usually need not memorize it. The ABI is enforced by the *toolchain*—the compiler, the linker, and so on—and does not typically otherwise surface. Knowledge of the ABI, however, can lead to more optimal programming and is required if writing assembly code or developing the toolchain itself (which is, after all, system programming).

The ABI is defined and implemented by the kernel and the toolchain.

Standards

Unix system programming is an old art. The basics of Unix programming have existed untouched for decades. Unix systems, however, are dynamic beasts. Behavior changes and features are added. To help bring order to chaos, standards groups codify system interfaces into official standards. Numerous such standards exist but, technically speaking, Linux does not officially comply with any of them. Instead, Linux *aims* toward compliance with two of the most important and prevalent standards: POSIX and the Single UNIX Specification (SUS).

POSIX and SUS document, among other things, the C API for a Unix-like operating system interface. Effectively, they define system programming, or at least a common subset thereof, for compliant Unix systems.

POSIX and SUS History

In the mid-1980s, the Institute of Electrical and Electronics Engineers (IEEE) spearheaded an effort to standardize system-level interfaces on Unix systems. Richard Stallman, founder of the Free Software movement, suggested the standard be named *POSIX* (pronounced *pahz-icks*), which now stands for *Portable Operating System Interface*.

The first result of this effort, issued in 1988, was IEEE Std 1003.1-1988 (POSIX 1988, for short). In 1990, the IEEE revised the POSIX standard with IEEE Std 1003.1-1990 (POSIX 1990). Optional real-time and threading support were documented in, respectively, IEEE Std 1003.1b-1993 (POSIX 1993 or POSIX.1b), and IEEE Std 1003.1c-1995 (POSIX 1995 or POSIX.1c). In 2001, the optional standards were rolled together with the base POSIX 1990, creating a single standard: IEEE Std 1003.1-2001 (POSIX 2001). The latest revision, released in December 2008, is IEEE Std 1003.1-2008 (POSIX 2008). All of the core POSIX standards are abbreviated POSIX.1, with the 2008 revision being the latest.

In the late 1980s and early 1990s, Unix system vendors were engaged in the "Unix Wars," with each struggling to define its Unix variant as *the* Unix operating system. Several major Unix vendors rallied around The Open Group, an industry consortium formed from the merging of the Open Software Foundation (OSF) and X/Open. The Open Group provides certification, white papers, and compliance testing. In the early 1990s, with the Unix Wars raging, The Open Group released the Single UNIX Specification (SUS). SUS rapidly grew in popularity, in large part due to its cost (free) versus the high cost of the POSIX standard. Today, SUS incorporates the latest POSIX standard.

The first SUS was published in 1994. This was followed by revisions in 1997 (SUSv2) and 2002 (SUSv3). The latest SUS, SUSv4, was published in 2008. SUSv4 revises and

combines IEEE Std 1003.1-2008 and several other standards. Throughout this book, I will mention when system calls and other interfaces are standardized by POSIX. I mention POSIX and not SUS because the latter subsumes the former.

C Language Standards

Dennis Ritchie and Brian Kernighan's famed book, *The C Programming Language* (Prentice Hall), acted as the informal C specification for many years following its 1978 publication. This version of C came to be known as *K&R C*. C was already rapidly replacing BASIC and other languages as the lingua franca of microcomputer programming. Therefore, to standardize the by-then quite popular language, in 1983 the American National Standards Institute (ANSI) formed a committee to develop an official version of C, incorporating features and improvements from various vendors and the new C++ language. The process was long and laborious, but *ANSI C* was completed in 1989. In 1990, the International Organization for Standardization (ISO) ratified *ISO C90*, based on ANSI C with a small handful of modifications.

In 1995, the ISO released an updated (although rarely implemented) version of the C language, *ISO C95*. This was followed in 1999 with a large update to the language, *ISO C99*, that introduced many new features, including inline functions, new data types, variable-length arrays, C++-style comments, and new library functions. The latest version of the standard is *ISO C11*, the most significant feature of which is a formalized memory model, enabling the portable use of threads across platforms.

On the C++ front, ISO standardization was slow in arriving. After years of development —and forward-incompatible compiler release—the first C `standard, ISO C`98, was ratified in 1998. While it greatly improved compatibility across compilers, several aspects of the standard limited consistency and portability. *ISO C++03* arrived in 2003. It offered bug fixes to aid compiler developers but no user-visible changes. The next and most recent ISO standard, *C++11* (formerly *C++0x* in suggestion of a more optimistic release date), heralded numerous language and standard library additions and improvements—so many, in fact, that many commentators suggest C++11 is a distinct language from previous C++ revisions.

Linux and the Standards

As stated earlier, Linux aims toward POSIX and SUS compliance. It provides the interfaces documented in SUSv4 and POSIX 2008, including real-time (POSIX.1b) and threading (POSIX.1c) support. More importantly, Linux strives to behave in accordance with POSIX and SUS requirements. In general, failing to agree with the standards is considered a bug. Linux is believed to comply with POSIX.1 and SUSv3, but as no official POSIX or SUS certification has been performed (particularly on each and every revision of Linux), we cannot say that Linux is officially POSIX- or SUS-compliant.

With respect to language standards, Linux fares well. The *gcc* C compiler is ISO C99-compliant; support for C11 is ongoing. The *g++* C++ compiler is ISO C++03-compliant with support for C++11 in development. In addition, *gcc* and *g++_* implement extensions to the C and C++ languages. These extensions are collectively called *GNU C*, and are documented in Appendix A.

Linux has not had a great history of forward compatibility,[1] although these days it fares much better. Interfaces documented by standards, such as the standard C library, will obviously always remain source compatible. Binary compatibility is maintained across a given major version of *glibc*, at the very least. And as C is standardized, *gcc* will always compile legal C correctly, although *gcc*-specific extensions may be deprecated and eventually removed with new *gcc* releases. Most importantly, the Linux kernel guarantees the stability of system calls. Once a system call is implemented in a stable version of the Linux kernel, it is set in stone.

Among the various Linux distributions, the Linux Standard Base (LSB) standardizes much of the Linux system. The LSB is a joint project of several Linux vendors under the auspices of the Linux Foundation (formerly the Free Standards Group). The LSB extends POSIX and SUS, and adds several standards of its own; it attempts to provide a binary standard, allowing object code to run unmodified on compliant systems. Most Linux vendors comply with the LSB to some degree.

This Book and the Standards

This book deliberately avoids paying lip service to any of the standards. Far too frequently, Unix system programming books must stop to elaborate on how an interface behaves in one standard versus another, whether a given system call is implemented on this system versus that, and similar page-filling bloat. This book, however, is specifically about system programming on a modern Linux system, as provided by the latest versions of the Linux kernel (3.9), gcc (4.8), and C library (2.17).

As system interfaces are generally set in stone—the Linux kernel developers go to great pains to never break the system call interfaces, for example—and provide some level of both source and binary compatibility, this approach allows us to dive into the details of Linux's system interface unfettered by concerns of compatibility with numerous other Unix systems and standards. This sole focus on Linux also enables this book to offer in-depth treatment of cutting-edge Linux-specific interfaces that will remain relevant and valid far into the future. The book draws upon an intimate knowledge of Linux and of the implementation and behavior of components such as *gcc* and the kernel, to provide an insider's view full of the best practices and optimization tips of an experienced veteran.

1. Experienced Linux users might remember the switch from *a.out* to ELF, the switch from *libc5* to *glibc*, *gcc* changes, *C++* template ABI breakages, and so on. Thankfully, those days are behind us.

Concepts of Linux Programming

This section presents a concise overview of the services provided by a Linux system. All Unix systems, Linux included, provide a mutual set of abstractions and interfaces. Indeed, these commonalities *define* Unix. Abstractions such as the file and the process, interfaces to manage pipes and sockets, and so on, are at the core of a Unix system.

This overview assumes that you are familiar with the Linux environment: I presume that you can get around in a shell, use basic commands, and compile a simple C program. This is *not* an overview of Linux or its programming environment, but rather of the foundation of Linux system programming.

Files and the Filesystem

The file is the most basic and fundamental abstraction in Linux. Linux follows the *everything-is-a-file* philosophy (although not as strictly as some other systems, such as Plan 9).[2] Consequently, much interaction occurs via reading of and writing to files, even when the object in question is not what you would consider a normal file.

In order to be accessed, a file must first be opened. Files can be opened for reading, writing, or both. An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself. Inside the Linux kernel, this descriptor is handled by an integer (of the C type `int`) called the *file descriptor*, abbreviated *fd*. File descriptors are shared with user space, and are used directly by user programs to access files. A large part of Linux system programming consists of opening, manipulating, closing, and otherwise using file descriptors.

Regular files

What most of us call "files" are what Linux labels *regular files*. A regular file contains bytes of data, organized into a linear array called a byte stream. In Linux, no further organization or formatting is specified for a file. The bytes may have any values, and they may be organized within the file in any way. At the system level, Linux does not enforce a structure upon files beyond the byte stream. Some operating systems, such as VMS, provide highly structured files, supporting concepts such as *records*. Linux does not.

Any of the bytes within a file may be read from or written to. These operations start at a specific byte, which is one's conceptual "location" within the file. This location is called the *file position* or *file offset*. The file position is an essential piece of the metadata that the kernel associates with each open file. When a file is first opened, the file position

2. Plan9, an operating system born of Bell Labs, is often called the successor to Unix. It features several innovative ideas, and is an adherent of the everything-is-a-file philosophy.

is zero. Usually, as bytes in the file are read from or written to, byte-by-byte, the file position increases in kind. The file position may also be set manually to a given value, even a value beyond the end of the file. Writing a byte to a file position beyond the end of the file will cause the intervening bytes to be padded with zeros. While it is possible to write bytes in this manner to a position beyond the end of the file, it is not possible to write bytes to a position before the beginning of a file. Such a practice sounds nonsensical, and, indeed, would have little use. The file position starts at zero; it cannot be negative. Writing a byte to the middle of a file overwrites the byte previously located at that offset. Thus, it is not possible to expand a file by writing into the middle of it. Most file writing occurs at the end of the file. The file position's maximum value is bounded only by the size of the C type used to store it, which is 64 bits on a modern Linux system.

The size of a file is measured in bytes and is called its *length*. The length, in other words, is simply the number of bytes in the linear array that make up the file. A file's length can be changed via an operation called *truncation*. A file can be truncated to a new size smaller than its original size, which results in bytes being removed from the end of the file. Confusingly, given the operation's name, a file can also be "truncated" to a new size larger than its original size. In that case, the new bytes (which are added to the end of the file) are filled with zeros. A file may be empty (that is, have a length of zero), and thus contain no valid bytes. The maximum file length, as with the maximum file position, is bounded only by limits on the sizes of the C types that the Linux kernel uses to manage files. Specific filesystems, however, may impose their own restrictions, imposing a smaller ceiling on the maximum length.

A single file can be opened more than once, by a different or even the same process. Each open instance of a file is given a unique file descriptor. Conversely, processes can share their file descriptors, allowing a single descriptor to be used by more than one process. The kernel does not impose any restrictions on concurrent file access. Multiple processes are free to read from and write to the same file at the same time. The results of such concurrent accesses rely on the ordering of the individual operations, and are generally unpredictable. User-space programs typically must coordinate amongst themselves to ensure that concurrent file accesses are properly synchronized.

Although files are usually accessed via *filenames*, they actually are not directly associated with such names. Instead, a file is referenced by an *inode* (originally short for *information node*), which is assigned an integer value unique to the filesystem (but not necessarily unique across the whole system). This value is called the *inode number*, often abbreviated as *i-number* or *ino*. An inode stores metadata associated with a file, such as its modification timestamp, owner, type, length, and the location of the file's data—but no filename! The inode is both a physical object, located on disk in Unix-style filesystems, and a conceptual entity, represented by a data structure in the Linux kernel.

Directories and links

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number. *Directories* are used to provide the names with which to access files. A directory acts as a mapping of human-readable names to inode numbers. A name and inode pair is called a *link*. The physical on-disk form of this mapping—for example, a simple table or a hash—is implemented and managed by the kernel code that supports a given filesystem. Conceptually, a directory is viewed like any normal file, with the difference that it contains only a mapping of names to inodes. The kernel directly uses this mapping to perform name-to-inode resolutions.

When a user-space application requests that a given filename be opened, the kernel opens the directory containing the filename and searches for the given name. From the filename, the kernel obtains the inode number. From the inode number, the inode is found. The inode contains metadata associated with the file, including the on-disk location of the file's data.

Initially, there is only one directory on the disk, the *root directory*. This directory is usually denoted by the path /. But, as we all know, there are typically many directories on a system. How does the kernel know *which* directory to look in to find a given filename?

As mentioned previously, directories are much like regular files. Indeed, they even have associated inodes. Consequently, the links inside of directories can point to the inodes of other directories. This means directories can nest inside of other directories, forming a hierarchy of directories. This, in turn, allows for the use of the *pathnames* with which all Unix users are familiar—for example, */home/blackbeard/concorde.png*.

When the kernel is asked to open a pathname like this, it walks each *directory entry* (called a *dentry* inside of the kernel) in the pathname to find the inode of the next entry. In the preceding example, the kernel starts at /, gets the inode for *home*, goes there, gets the inode for *blackbeard*, runs there, and finally gets the inode for *concorde.png*. This operation is called *directory* or *pathname resolution*. The Linux kernel also employs a cache, called the *dentry cache*, to store the results of directory resolutions, providing for speedier lookups in the future given temporal locality.[3]

A pathname that starts at the root directory is said to be *fully qualified*, and is called an *absolute pathname*. Some pathnames are not fully qualified; instead, they are provided relative to some other directory (for example, *todo/plunder*). These paths are called *relative pathnames*. When provided with a relative pathname, the kernel begins the pathname resolution in the *current working directory*. From the current working

3. *Temporal locality* is the high likelihood of an access to a particular resource being followed by another access to the same resource. Many resources on a computer exhibit temporal locality.

directory, the kernel looks up the directory *todo*. From there, the kernel gets the inode for *plunder*. Together, the combination of a relative pathname and the current working directory is fully qualified.

Although directories are treated like normal files, the kernel does not allow them to be opened and manipulated like regular files. Instead, they must be manipulated using a special set of system calls. These system calls allow for the adding and removing of links, which are the only two sensible operations anyhow. If user space were allowed to manipulate directories without the kernel's mediation, it would be too easy for a single simple error to corrupt the filesystem.

Hard links

Conceptually, nothing covered thus far would prevent multiple names resolving to the same inode. Indeed, this is allowed. When multiple links map different names to the same inode, we call them *hard links*.

Hard links allow for complex filesystem structures with multiple pathnames pointing to the same data. The hard links can be in the same directory, or in two or more different directories. In either case, the kernel simply resolves the pathname to the correct inode. For example, a specific inode that points to a specific chunk of data can be hard linked from */home/bluebeard/treasure.txt* and */home/blackbeard/to_steal.txt*.

Deleting a file involves *unlinking* it from the directory structure, which is done simply by removing its name and inode pair from a directory. Because Linux supports hard links, however, the filesystem cannot destroy the inode and its associated data on every unlink operation. What if another hard link existed elsewhere in the filesystem? To ensure that a file is not destroyed until *all* links to it are removed, each inode contains a *link count* that keeps track of the number of links within the filesystem that point to it. When a pathname is unlinked, the link count is decremented by one; only when it reaches zero are the inode and its associated data actually removed from the filesystem.

Symbolic links

Hard links cannot span filesystems because an inode number is meaningless outside of the inode's own filesystem. To allow links that can span filesystems, and that are a bit simpler and less transparent, Unix systems also implement *symbolic links* (often shortened to *symlinks*).

Symbolic links look like regular files. A symlink has its own inode and data chunk, which contains the complete pathname of the linked-to file. This means symbolic links can point anywhere, including to files and directories that reside on different filesystems, and even to files and directories that do not exist. A symbolic link that points to a nonexistent file is called a *broken link*.

Symbolic links incur more overhead than hard links because resolving a symbolic link effectively involves resolving two files: the symbolic link and then the linked-to file. Hard links do not incur this additional overhead—there is no difference between accessing a file linked into the filesystem more than once and one linked only once. The overhead of symbolic links is minimal, but it is still considered a negative.

Symbolic links are also more opaque than hard links. Using hard links is entirely transparent; in fact, it takes effort to find out that a file is linked more than once! Manipulating symbolic links, on the other hand, requires special system calls. This lack of transparency is often considered a positive, as the link structure is explicitly made plain, with symbolic links acting more as *shortcuts* than as filesystem-internal links.

Special files

Special files are kernel objects that are represented as files. Over the years, Unix systems have supported a handful of different special files. Linux supports four: block device files, character device files, named pipes, and Unix domain sockets. Special files are a way to let certain abstractions fit into the filesystem, continuing the everything-is-a-file paradigm. Linux provides a system call to create a special file.

Device access in Unix systems is performed via device files, which act and look like normal files residing on the filesystem. Device files may be opened, read from, and written to, allowing user space to access and manipulate devices (both physical and virtual) on the system. Unix devices are generally broken into two groups: *character devices* and *block devices*. Each type of device has its own special device file.

A character device is accessed as a linear queue of bytes. The device driver places bytes onto the queue, one by one, and user space reads the bytes in the order that they were placed on the queue. A keyboard is an example of a character device. If the user types "peg," for example, an application would want to read from the keyboard device the *p*, the *e*, and, finally, the *g*, in exactly that order. When there are no more characters left to read, the device returns end-of-file (EOF). Missing a character, or reading them in any other order, would make little sense. Character devices are accessed via *character device files*.

A block device, in contrast, is accessed as an array of bytes. The device driver maps the bytes over a seekable device, and user space is free to access any valid bytes in the array, in any order—it might read byte 12, then byte 7, and then byte 12 again. Block devices are generally storage devices. Hard disks, floppy drives, CD-ROM drives, and flash memory are all examples of block devices. They are accessed via *block device files*.

Named pipes (often called *FIFOs*, short for "first in, first out") are an *interprocess communication* (*IPC*) mechanism that provides a communication channel over a file descriptor, accessed via a special file. Regular pipes are the method used to "pipe" the output of one program into the input of another; they are created in memory via a system call and do not exist on any filesystem. Named pipes act like regular pipes but

are accessed via a file, called a *FIFO special file*. Unrelated processes can access this file and communicate.

Sockets are the final type of special file. Sockets are an advanced form of IPC that allow for communication between two different processes, not only on the same machine, but even on two different machines. In fact, sockets form the basis of network and Internet programming. They come in multiple varieties, including the Unix domain socket, which is the form of socket used for communication within the local machine. Whereas sockets communicating over the Internet might use a hostname and port pair for identifying the target of communication, Unix domain sockets use a special file residing on a filesystem, often simply called a socket file.

Filesystems and namespaces

Linux, like all Unix systems, provides a global and unified *namespace* of files and directories. Some operating systems separate different disks and drives into separate namespaces—for example, a file on a floppy disk might be accessible via the pathname *A:\plank.jpg*, while the hard drive is located at *C:*. In Unix, that same file on a floppy might be accessible via the pathname */media/floppy/plank.jpg* or even via */home/captain/stuff/plank.jpg*, right alongside files from other media. That is, on Unix, the namespace is unified.

A *filesystem* is a collection of files and directories in a formal and valid hierarchy. Filesystems may be individually added to and removed from the global namespace of files and directories. These operations are called *mounting* and *unmounting*. Each filesystem is mounted to a specific location in the namespace, known as a *mount point*. The root directory of the filesystem is then accessible at this mount point. For example, a CD might be mounted at */media/cdrom*, making the root of the filesystem on the CD accessible at */media/cdrom*. The first filesystem mounted is located in the root of the namespace, /, and is called the *root filesystem*. Linux systems always have a root filesystem. Mounting other filesystems at other mount points is optional.

Filesystems usually exist physically (i.e., are stored on disk), although Linux also supports *virtual filesystems* that exist only in memory, and *network filesystems* that exist on machines across the network. Physical filesystems reside on block storage devices, such as CDs, floppy disks, compact flash cards, or hard drives. Some such devices are *partionable*, which means that they can be divided up into multiple filesystems, all of which can be manipulated individually. Linux supports a wide range of filesystems—certainly anything that the average user might hope to come across—including media-specific filesystems (for example, ISO9660), network filesystems (*NFS*), native filesystems (*ext4*), filesystems from other Unix systems (*XFS*), and even filesystems from non-Unix systems (*FAT*).

The smallest addressable unit on a block device is the *sector*. The sector is a physical attribute of the device. Sectors come in various powers of two, with 512 bytes being

quite common. A block device cannot transfer or access a unit of data smaller than a sector and all I/O must occur in terms of one or more sectors.

Likewise, the smallest logically addressable unit on a filesystem is the *block*. The block is an abstraction of the filesystem, not of the physical media on which the filesystem resides. A block is usually a power-of-two multiple of the sector size. In Linux, blocks are generally larger than the sector, but they must be smaller than the *page size* (the smallest unit addressable by the *memory management unit*, a hardware component).[4] Common block sizes are 512 bytes, 1 kilobyte, and 4 kilobytes.

Historically, Unix systems have only a single shared namespace, viewable by all users and all processes on the system. Linux takes an innovative approach and supports *per-process namespaces*, allowing each process to optionally have a unique view of the system's file and directory hierarchy.[5] By default, each process inherits the namespace of its parent, but a process may elect to create its own namespace with its own set of mount points and a unique root directory.

Processes

If files are the most fundamental abstraction in a Unix system, processes are the runner up. Processes are object code in execution: active, running programs. But they're more than just object code—processes consist of data, resources, state, and a virtualized computer.

Processes begin life as executable object code, which is machine-runnable code in an executable format that the kernel understands. The format most common in Linux is called *Executable and Linkable Format* (*ELF*). The executable format contains metadata, and multiple *sections* of code and data. Sections are linear chunks of the object code that load into linear chunks of memory. All bytes in a section are treated the same, given the same permissions, and generally used for similar purposes.

The most important and common sections are the *text section*, the *data section*, and the *bss section*. The text section contains executable code and read-only data, such as constant variables, and is typically marked read-only and executable. The data section contains initialized data, such as C variables with defined values, and is typically marked readable and writable. The bss section contains uninitialized global data. Because the C standard dictates default values for global C variables that are essentially all zeros, there is no need to store the zeros in the object code on disk. Instead, the object code can simply list the uninitialized variables in the bss section, and the kernel can map the *zero page* (a page of all zeros) over the section when it is loaded into memory. The bss section was conceived solely as an optimization for this purpose. The name is a historic relic;

4. This is an artificial kernel limitation in the name of simplicity which may go away in the future.

5. This approach was first pioneered by Bell Labs' Plan 9.

it stands for *block started by symbol*. Other common sections in ELF executables are the *absolute section* (which contains nonrelocatable symbols) and the *undefined section* (a catchall).

A process is also associated with various system resources, which are arbitrated and managed by the kernel. Processes typically request and manipulate resources only through system calls. Resources include timers, pending signals, open files, network connections, hardware, and IPC mechanisms. A process's resources, along with data and statistics related to the process, are stored inside the kernel in the process's *process descriptor*.

A process is a virtualization abstraction. The Linux kernel, supporting both preemptive multitasking and virtual memory, provides every process both a virtualized processor and a virtualized view of memory. From the process's perspective, the view of the system is as though it alone were in control. That is, even though a given process may be scheduled alongside many other processes, it runs as though it has sole control of the system. The kernel seamlessly and transparently preempts and reschedules processes, sharing the system's processors among all running processes. Processes never know the difference. Similarly, each process is afforded a single linear address space, as if it alone were in control of all of the memory in the system. Through virtual memory and paging, the kernel allows many processes to coexist on the system, each operating in a different address space. The kernel manages this virtualization through hardware support provided by modern processors, allowing the operating system to concurrently manage the state of multiple independent processes.

Threads

Each process consists of one or more *threads of execution* (usually simplified to *threads*). A thread is the unit of activity within a process. In other words, a thread is the abstraction responsible for executing code and maintaining the process's running state.

Most processes consist of only a single thread; they are called *single-threaded*. Processes that contain multiple threads are said to be *multithreaded*. Traditionally, Unix programs have been single-threaded, owing to Unix's historic simplicity, fast process creation times, and robust IPC mechanisms, all of which mitigate the desire for threads.

A thread consists of a *stack* (which stores its local variables, just as the process stack does on nonthreaded systems), processor state, and a current location in the object code (usually stored in the processor's *instruction pointer*). The majority of the remaining parts of a process are shared among all threads, most notably the process address space. In this manner, threads share the virtual memory abstraction while maintaining the virtualized processor abstraction.

Internally, the Linux kernel implements a unique view of threads: they are simply normal processes that happen to share some resources. In user space, Linux implements threads in accordance with POSIX 1003.1c (known as *Pthreads*). The name of the current Linux

thread implementation, which is part of *glibc*, is the *Native POSIX Threading Library* (*NPTL*). We'll discuss threads more in Chapter 7.

Process hierarchy

Each process is identified by a unique positive integer called the *process ID* (pid). The pid of the first process is 1, and each subsequent process receives a new, unique pid.

In Linux, processes form a strict hierarchy, known as the *process tree*. The process tree is rooted at the first process, known as the *init process*, which is typically the *init* program. New processes are created via the `fork()` system call. This system call creates a duplicate of the calling process. The original process is called the *parent*; the new process is called the *child*. Every process except the first has a parent. If a parent process terminates before its child, the kernel will *reparent* the child to the init process.

When a process terminates, it is not immediately removed from the system. Instead, the kernel keeps parts of the process resident in memory to allow the process's parent to inquire about its status upon terminating. This inquiry is known as *waiting on* the terminated process. Once the parent process has waited on its terminated child, the child is fully destroyed. A process that has terminated, but has not yet been waited upon, is called a *zombie*. The init process routinely waits on all of its children, ensuring that reparented processes do not remain zombies forever.

Users and Groups

Authorization in Linux is provided by *users* and *groups*. Each user is associated with a unique positive integer called the *user ID* (uid). Each process is in turn associated with exactly one uid, which identifies the user running the process, and is called the process's *real uid*. Inside the Linux kernel, the uid is the only concept of a user. Users, however, refer to themselves and other users through *usernames*, not numerical values. Usernames and their corresponding uids are stored in */etc/passwd*, and library routines map user-supplied usernames to the corresponding uids.

During login, the user provides a username and password to the *login* program. If given a valid username and the correct password, the *login* program spawns the user's *login shell*, which is also specified in */etc/passwd*, and makes the shell's uid equal to that of the user. Child processes inherit the uids of their parents.

The uid 0 is associated with a special user known as *root*. The root user has special privileges, and can do almost anything on the system. For example, only the root user can change a process's uid. Consequently, the *login* program runs as root.

In addition to the real uid, each process also has an *effective uid*, a *saved uid*, and a *filesystem uid*. While the real uid is always that of the user who started the process, the effective uid may change under various rules to allow a process to execute with the rights of different users. The saved uid stores the original effective uid; its value is used in

deciding what effective uid values the user may switch to. The filesystem uid, which is usually equal to the effective uid, is used for verifying filesystem access.

Each user belongs to one or more groups, including a *primary* or *login group*, listed in */etc/passwd*, and possibly a number of *supplemental groups*, listed in */etc/group*. Each process is therefore also associated with a corresponding *group ID* (gid), and has a *real gid*, an *effective gid*, a *saved gid*, and a *filesystem gid*. Processes are generally associated with a user's login group, not any of the supplemental groups.

Certain security checks allow processes to perform certain operations only if they meet specific criteria. Historically, Unix has made this decision very black-and-white: processes with uid 0 had access, while no others did. Recently, Linux has replaced this security system with a more general *capabilities* system. Instead of a simple binary check, capabilities allow the kernel to base access on much more fine-grained settings.

Permissions

The standard file permission and security mechanism in Linux is the same as that in historic Unix.

Each file is associated with an owning user, an owning group, and three sets of permission bits. The bits describe the ability of the owning user, the owning group, and everybody else to read, write, and execute the file; there are three bits for each of the three classes, making nine bits in total. The owners and the permissions are stored in the file's inode.

For regular files, the permissions are rather obvious: they specify the ability to open a file for reading, open a file for writing, or execute a file. Read and write permissions are the same for special files as for regular files, although what exactly is read or written is up to the special file in question. Execute permissions are ignored on special files. For directories, read permission allows the contents of the directory to be listed, write permission allows new links to be added inside the directory, and execute permission allows the directory to be entered and used in a pathname. Table 1-1 lists each of the nine permission bits, their octal values (a popular way of representing the nine bits), their text values (as *ls* might show them), and their corresponding meanings.

Table 1-1. Permission bits and their values

Bit	Octal value	Text value	Corresponding permission
8	400	`r--------`	Owner may read
7	200	`-w-------`	Owner may write
6	100	`--x------`	Owner may execute
5	040	`---r-----`	Group may read
4	020	`----w----`	Group may write
3	010	`-----x---`	Group may execute

Bit	Octal value	Text value	Corresponding permission
2	004	`------r--`	Everyone else may read
1	002	`-------w-`	Everyone else may write
0	001	`--------x`	Everyone else may execute

In addition to historic Unix permissions, Linux also supports access control lists (ACLs). ACLs allow for much more detailed and exacting permission and security controls, at the cost of increased complexity and on-disk storage.

Signals

Signals are a mechanism for one-way asynchronous notifications. A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself. Signals typically alert a process to some event, such as a segmentation fault or the user pressing Ctrl-C.

The Linux kernel implements about 30 signals (the exact number is architecture-dependent). Each signal is represented by a numeric constant and a textual name. For example, `SIGHUP`, used to signal that a terminal hangup has occurred, has a value of `1` on the x86-64 architecture.

Signals *interrupt* an executing process, causing it to stop whatever it is doing and immediately perform a predetermined action. With the exception of `SIGKILL` (which always terminates the process), and `SIGSTOP` (which always stops the process), processes may control what happens when they receive a signal. They can accept the default action, which may be to terminate the process, terminate and coredump the process, stop the process, or do nothing, depending on the signal. Alternatively, processes can elect to explicitly ignore or handle signals. Ignored signals are silently dropped. Handled signals cause the execution of a user-supplied *signal handler* function. The program jumps to this function as soon as the signal is received. When the signal handler returns, the control of the program resumes at the previously interrupted instruction. Because of the asynchrony of signals, signal handlers must take care not to stomp on the code that was interrupted, by executing only *async-safe* (also called *signal-safe*) functions.

Interprocess Communication

Allowing processes to exchange information and notify each other of events is one of an operating system's most important jobs. The Linux kernel implements most of the historic Unix IPC mechanisms—including those defined and standardized by both System V and POSIX—as well as implementing a mechanism or two of its own.

IPC mechanisms supported by Linux include pipes, named pipes, semaphores, message queues, shared memory, and futexes.

Headers

Linux system programming revolves around a handful of headers. Both the kernel itself and *glibc* provide the headers used in system-level programming. These headers include the standard C fare (for example, `<string.h>`), and the usual Unix offerings (say, `<unistd.h>`).

Error Handling

It goes without saying that checking for and handling errors are of paramount importance. In system programming, an error is signified via a function's return value and described via a special variable, `errno`. *glibc* transparently provides `errno` support for both library and system calls. The vast majority of interfaces covered in this book will use this mechanism to communicate errors.

Functions notify the caller of errors via a special return value, which is usually `-1` (the exact value used depends on the function). The error value alerts the caller to the occurrence of an error but provides no insight into why the error occurred. The `errno` variable is used to find the cause of the error.

This variable is declared in `<errno.h>` as follows:

```
extern int errno;
```

Its value is valid only immediately after an `errno`-setting function indicates an error (usually by returning `-1`), as it is legal for the variable to be modified during the successful execution of a function.

The `errno` variable may be read or written directly; it is a modifiable lvalue. The value of `errno` maps to the textual description of a specific error. A preprocessor `#define` also maps to the numeric `errno` value. For example, the preprocessor define `EACCES` equals `1`, and represents "permission denied." See Table 1-2 for a listing of the standard defines and the matching error descriptions.

Table 1-2. Errors and their descriptions

Preprocessor define	Description
`E2BIG`	Argument list too long
`EACCES`	Permission denied
`EAGAIN`	Try again
`EBADF`	Bad file number
`EBUSY`	Device or resource busy
`ECHILD`	No child processes
`EDOM`	Math argument outside of domain of function
`EEXIST`	File already exists

Preprocessor define	Description
EFAULT	Bad address
EFBIG	File too large
EINTR	System call was interrupted
EINVAL	Invalid argument
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENFILE	File table overflow
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOMEM	Out of memory
ENOSPC	No space left on device
ENOTDIR	Not a directory
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Result too large
EROFS	Read-only filesystem
ESPIPE	Invalid seek
ESRCH	No such process
ETXTBSY	Text file busy
EXDEV	Improper link

The C library provides a handful of functions for translating an `errno` value to the corresponding textual representation. This is needed only for error reporting, and the like; checking and handling errors can be done using the preprocessor defines and `errno` directly.

The first such function is `perror()`:

```
#include <stdio.h>

void perror (const char *str);
```

This function prints to *stderr* (standard error) the string representation of the current error described by `errno`, prefixed by the string pointed at by `str`, followed by a colon. To be useful, the name of the function that failed should be included in the string. For example:

```
if (close (fd) == -1)
        perror ("close");
```

The C library also provides `strerror()` and `strerror_r()`, prototyped as:

```
#include <string.h>

char * strerror (int errnum);
```

and:

```
#include <string.h>

int strerror_r (int errnum, char *buf, size_t len);
```

The former function returns a pointer to a string describing the error given by `errnum`. The string may not be modified by the application but can be modified by subsequent `perror()` and `strerror()` calls. Thus, it is not thread-safe.

The `strerror_r()` function is thread-safe. It fills the buffer of length `len` pointed at by `buf`. A call to `strerror_r()` returns `0` on success, and `-1` on failure. Humorously, it sets `errno` on error.

For a few functions, the entire range of the return type is a legal return value. In those cases, `errno` must be zeroed before invocation and checked afterward (these functions promise to only return a nonzero `errno` on actual error). For example:

```
errno = 0;
arg = strtoul (buf, NULL, 0);
if (errno)
        perror ("strtoul");
```

A common mistake in checking `errno` is to forget that any library or system call can modify it. For example, this code is buggy:

```
if (fsync (fd) == -1) {
        fprintf (stderr, "fsync failed!\n");
        if (errno == EIO)
                fprintf (stderr, "I/O error on %d!\n", fd);
}
```

If you need to preserve the value of `errno` across function invocations, save it:

```
if (fsync (fd) == -1) {
        const int err = errno;
        fprintf (stderr, "fsync failed: %s\n", strerror (errno));
        if (err == EIO) {
                /* if the error is I/O-related, jump ship */
                fprintf (stderr, "I/O error on %d!\n", fd);
                exit (EXIT_FAILURE);
        }
}
```

In single-threaded programs, `errno` is a global variable, as shown earlier in this section. In multithreaded programs, however, `errno` is stored per-thread, and is thus thread-safe.

Getting Started with System Programming

This chapter looked at the fundamentals of Linux system programming and provided a programmer's overview of the Linux system. The next chapter discusses basic file I/O. This includes, of course, reading from and writing to files; however, because Linux implements many interfaces as files, file I/O is crucial to a lot more than just, well, files.

With the preliminaries behind us, it is time to dive into actual system programming. Let's go!