DOWNLOAD PDF LINUX SYSTEM PROGRAMMING 2ND EDITION

Chapter 1 : LINUX SYSTEM PROGRAMMING, 2/E By ROBERT LOVE

Write software that draws directly on services offered by the Linux kernel and core system libraries. With this comprehensive book, Linux kernel contributor Robert Love provides you with a tutorial on Linux system programming, a reference manual on Linux system calls, and an insider's guide to.

With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, tutorials, and more. 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. 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 itselfâ€”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. 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 â€”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. 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: 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. This is traditional system programmingâ€”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: 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 spaceâ€”your text editor, favorite game, and so onâ€”into the kernel the core internals of the system in order to request some service or resource from the operating system. Linux implements far fewer system calls than most other operating system kernels. In the Linux kernel, each machine architecture such as Alpha, x, 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 callsâ€”more than 90 percentâ€”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. 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 i, for example, a user-space application executes a software interrupt instruction, int, with a value of 0x This instruction causes a switch into kernel space, the protected realm of the kernel, where the kernel executes a software interrupt handlerâ€”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 i 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 i, for example, a register is used for each possible parameterâ€”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. 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. 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. 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. Both define and describe the interfaces between different pieces of computer software. It provides abstraction by providing a standard set of interfacesâ€”â€”usually functionsâ€”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. It may use the API as-is, or not use it at all: 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. 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 i on Unix systems , the efforts have not met with much success. Instead, operating systemsâ€”Linux includedâ€”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 x Thus, the ABI is a function of both the operating system say, Linux and the architecture say, x System programmers ought to be aware of the ABI but usually need not memorize it.

Chapter 2 : Linux System Programming, 2nd Edition [Book]

Q&A with Robert Love, author of "Linux System Programming, 2nd Edition" Q. Why is your book timely-- what makes it important right now? A. Developing system software on Unix has always been in vogue, but we've seen a large increase in demand with the rise of the cloud and the web.

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. In particular, this book is about system programming on Linux. Linux is a modern Unixlike 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 itselfâ€”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. But note that this book does not cover the Linux programming 1 environment â€”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. These libraries abstract away the details of the hardware and operating system. Such abstraction has several goals: 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 higherlevel 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. 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. The umbrella of system programming often includes kernel development, or at least device driver writing. Device driver writing is a large, expansive topic, best tackled in books dedicated to the subject. Introduction and Essential Concepts 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: 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 spaceâ€”your text editor, favorite game, and so onâ€”into the kernel the core internals of the system in order to request some service or resource from the operating system. Linux implements far fewer system calls than most other operating system kernels. In the Linux kernel, each machine architecture such as Alpha, x, 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 callsâ€”more than 90 percentâ€”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. 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. This instruction causes a switch into kernel space, the protected realm of the kernel, where the kernel executes a software interrupt handlerâ€”and what is the handler for interrupt 0x80? None other than the system call handler! System Programming 3 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 i 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 i, for example, a register is used for each possible parameterâ€”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. 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. 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. At the system level, there are two separate sets of definitions and descriptions that impact portability. It provides abstraction by providing a standard set of interfacesâ€”â€”usually functionsâ€”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. It may use the API as-is, or not use it at all: 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. Instead, operating systemsâ€”Linux includedâ€” 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 x Thus, the ABI is a function of both the operating system say, Linux and the architecture say, x System programmers ought to be aware of the ABI but usually need not memorize it. The ABI is enforced by the toolchainâ€”the compiler, the linker, and so onâ€”and does not typically otherwise surface. 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. Instead, Linux aims toward compliance with two of the most important and prevalent standards: Effectively, they define system programming, or at least a common subset thereof, for compliant Unix systems. In , the optional standards were rolled together with the base POSIX , creating a single standard: The Open Group provides certification, white papers, and compliance testing. The first SUS was published in After years of development â€”and forward-incompatible compiler releaseâ€”the first C standard, ISO C98, was ratified in It offered bug fixes to aid compiler developers but no user-visible changes. In general, failing to agree with the standards is considered a bug. Introduction and Essential Concepts With

respect to language standards, Linux fares well. 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. 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.

Chapter 3 : Linux System Programming, 2nd Edition: Robert Love - IT eBooks - pdf

Stay ahead with the world's most comprehensive technology and business learning platform. With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, tutorials, and more.

Chapter 4 : Linux System Programming, 2nd Edition - theinnatdunvilla.com - theinnatdunvilla.com

Linux System Programming: Talking Directly to the Kernel and C Library Write software that draws directly on services offered by the Linux kernel and core system libraries. With this comprehensive book, Linux kernel contributor Robert Love provides you with a tutorial on Linux system programming, a reference manual on Linux system calls, and an.

Chapter 5 : Learning Linux Shell Scripting - Second Edition | PACKT Books

Book Description Write software that draws directly on services offered by the Linux kernel and core system libraries. With this comprehensive book, Linux kernel contributor Robert Love provides you with a tutorial on Linux system programming, a reference manual on Linux system calls, and an insider's guide to writing smarter, faster code.

Chapter 6 : Linux System Programming, 2nd Edition - PDF Free Download - Fox eBook

Linux System Programming. This book is about writing software that makes the most effective use of the system you're running on - code that interfaces directly with the kernel and core system libraries, including the shell, text editor, compiler, debugger, core utilities, and system daemons.

Chapter 7 : Linux System Programming, 2nd Edition : Books

Linux System Programming, 2nd Edition: Talking Directly to the Kernel and C Library Write software that draws directly on services offered by the Linux kernel and core system libraries. With this comprehensive book, Linux kernel contributor Robert Love provides you with a tutorial on Linux system programming, a reference manual on Linux system.

Chapter 8 : Linux System Programming, 2nd Edition - eBookee

Rent Linux System Programming 2nd edition () today, or search our site for other textbooks by Robert Love. Every textbook comes with a day "Any Reason" guarantee. Published by O'Reilly Media, Incorporated.

Chapter 9 : Mastering Linux Shell Scripting - Second Edition | PACKT Books

With this comprehensive book, Linux kernel contributor Robert Love provides you with a tutorial on Linux system programming, a reference manual on Linux system calls, and an insider's guide to writing smarter, faster code.