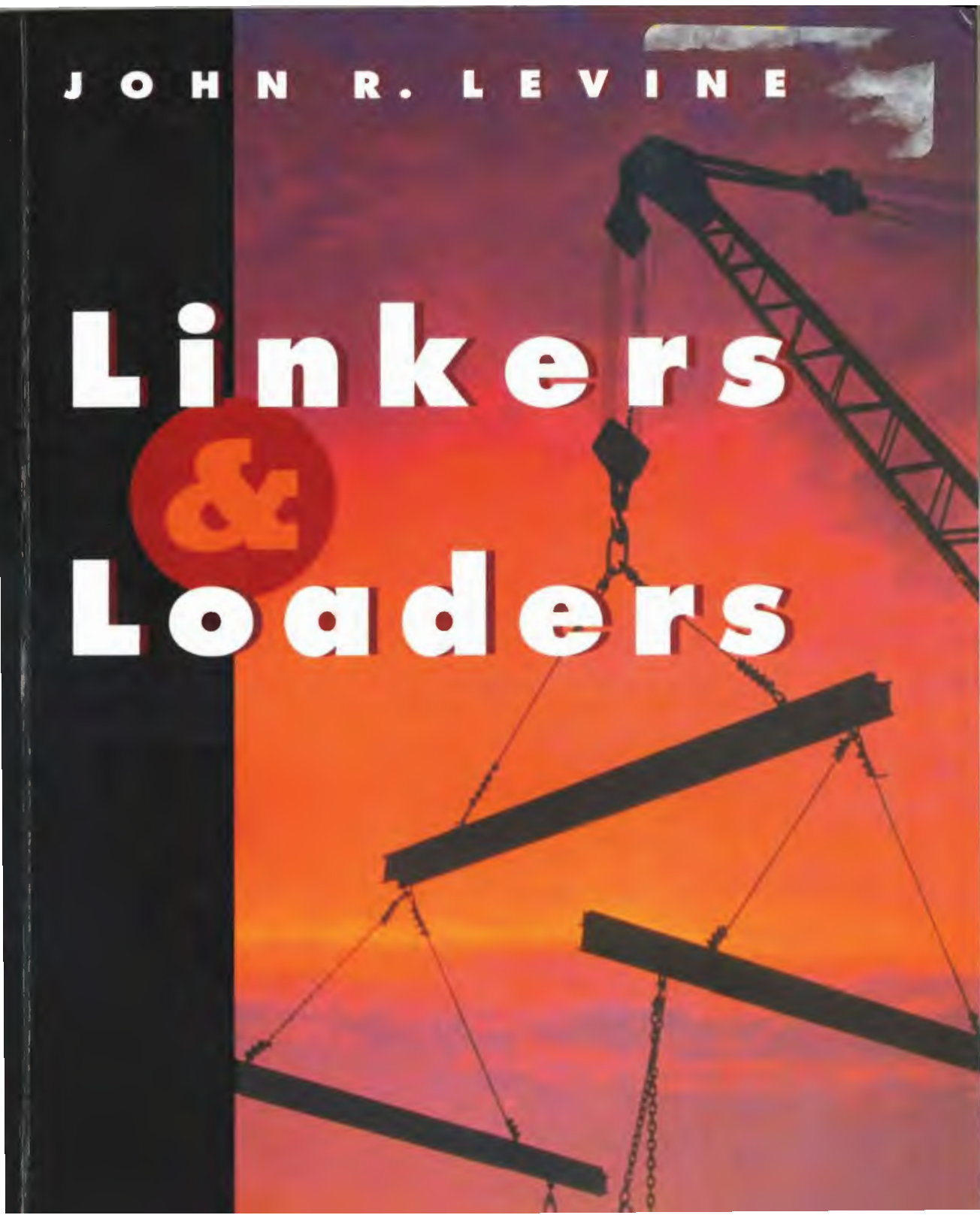


JOHN R. LEVINE

Linkers & Loaders



LINKERS AND LOADERS

LINKERS AND LOADERS

John R. Levine



MORGAN KAUFMANN PUBLISHERS

An Imprint of Elsevier

SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Acquisitions Editor Tim Cox
Director of Production and Manufacturing Yonie Overton
Production Editor Edward Wade
Editorial Assistant Sarah Luger
Cover Design Ross Carron Design
Cover Photograph © Greg L. Ryan; Sally Beyer/AllStock/PNI
Text Design, Technical Illustration, and Composition Rebecca Evans & Associates
Copyeditor Jeff Van Bueren
Proofreader Jennifer McClain
Indexer Bruce Tracy
Printer Courier Corporation

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances where Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

ACADEMIC PRESS

An Imprint of Elsevier

525 B Street, Suite 1900, San Diego, CA 92101-4495, USA

<http://www.academicpress.com>

Academic Press

24–28 Oval Road, London, NW1 7DX United Kingdom

<http://www.hbuk.co.uk/ap/>

Morgan Kaufmann Publishers

340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205

<http://www.mkp.com>

© 2000 by Academic Press

All rights reserved

Printed in the United States of America

Transferred to Digital Printing, 2010

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, or otherwise—without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science and Technology Rights Department in Oxford, UK. Phone: (44) 1865 843830, Fax: (44) 1865 853333, e-mail: permissions@elsevier.co.uk. You may also complete your request on-line via the Elsevier homepage: <http://www.elsevier.com> by selecting "Customer Support" and then "Obtaining Permissions".

Library of Congress Cataloging-in-Publication Data

Levine, John R.

Linkers and loaders / John R. Levine.

p. cm.

ISBN-13: 978-1-55860-496-4 ISBN-10: 1-55860-496-0

1. Assembling (Electronic computers) 2. Linkers (Computer programs) 3. Loaders (Computer programs) I. Title.

QA76.76.A87 L48 2000

005.4'56—dc21

99-047127

To Tonia and Sarah

CONTENTS

PREFACE	xi
----------------------	-----------

1	LINKING AND LOADING	1
----------	----------------------------------	----------

- 1.1 What Do Linkers and Loaders Do? 1
- 1.2 Address Binding: A Historical Perspective 1
- 1.3 Linking vs. Loading 5
- 1.4 Compiler Drivers 10
- 1.5 Linking: A True-Life Example 12

Exercises 17

2	ARCHITECTURAL ISSUES	19
----------	-----------------------------------	-----------

- 2.1 Application Binary Interfaces 20
- 2.2 Memory Addresses 20
- 2.3 Address Formation 22
- 2.4 Instruction Formats 23
- 2.5 Procedure Calls and Addressability 24
- 2.6 Data and Instruction References 28
- 2.7 Paging and Virtual Memory 34
- 2.8 Intel 386 Segmentation 41
- 2.9 Embedded Architectures 43

Exercises 45

3 OBJECT FILES 47

- 3.1 What Goes into an Object File? 47
- 3.2 The Null Object Format: DOS COM Files 49
- 3.3 Code Sections: UNIX a.out Files 50
- 3.4 Relocation: DOS EXE Files 56
- 3.5 Symbols and Relocation 58
- 3.6 Relocatable a.out 58
- 3.7 UNIX ELF 62
- 3.8 IBM 360 Object Format 71
- 3.9 Microsoft Portable Executable Format 75
- 3.10 Intel/Microsoft OMF Files 84
- 3.11 Comparison of Object Formats 90

Exercises 90

Project 90

4 STORAGE ALLOCATION 93

- 4.1 Segments and Addresses 93
- 4.2 Simple Storage Layout 94
- 4.3 Multiple-Segment Types 96
- 4.4 Segment and Page Alignment 96
- 4.5 Common Blocks and Other Special Segments 98
- 4.6 Linker Control Scripts 107
- 4.7 Storage Allocation in Practice 108

Exercises 114

Project 115

5 SYMBOL MANAGEMENT 117

- 5.1 Binding and Name Resolution 117
- 5.2 Symbol Table Formats 118
- 5.3 Name Mangling 124
- 5.4 Weak External and Other Kinds of Symbols 129
- 5.5 Maintaining Debugging Information 129

Exercises 132

Project 133

6 LIBRARIES 135

- 6.1 Purpose of Libraries 135
- 6.2 Library Formats 136
- 6.3 Creating Libraries 141
- 6.4 Searching Libraries 142
- 6.5 Performance Issues 144
- 6.6 Weak External Symbols 144

Exercises 145

Project 146

7 RELOCATION 149

- 7.1 Hardware and Software Relocation 150
- 7.2 Link-Time and Load-Time Relocation 151
- 7.3 Symbol and Segment Relocation 151
- 7.4 Basic Relocation Techniques 153
- 7.5 Relinkable and Relocatable Output Formats 159
- 7.6 Other Relocation Formats 160
- 7.7 Relocation Special Cases 162

Exercises 163

Project 164

8 LOADING AND OVERLAYS 167

- 8.1 Basic Loading 167
- 8.2 Basic Loading, with Relocation 168
- 8.3 Position-Independent Code 169
- 8.4 Bootstrap Loading 176
- 8.5 Tree-Structured Overlays 177

Exercises 184

Project 185

9 SHARED LIBRARIES 187

- 9.1 Binding Time 188
- 9.2 Shared Libraries in Practice 190
- 9.3 Address Space Management 190
- 9.4 Structure of Shared Libraries 192
- 9.5 Creating Shared Libraries 192

9.6	Linking with Shared Libraries	197
9.7	Running with Shared Libraries	197
9.8	The malloc Hack and Other Shared-Library Problems	199
	Exercises	202
	Project	203

10 DYNAMIC LINKING AND LOADING 205

10.1	ELF Dynamic Linking	206
10.2	Contents of an ELF File	206
10.3	Loading a Dynamically Linked Program	210
10.4	Lazy Procedure Linkage with the PLT	213
10.5	Other Peculiarities of Dynamic Linking	215
10.6	Dynamic Loading at Run Time	216
10.7	Microsoft Dynamic-Link Libraries	217
10.8	OSF/1 Pseudo-Static Shared Libraries	222
10.9	Making Shared Libraries Fast	223
10.10	Comparison of Dynamic Linking Approaches	224
	Exercises	226
	Project	226

11 ADVANCED TECHNIQUES 229

11.1	Techniques for C++	229
11.2	Incremental Linking and Relinking	233
11.3	Link-Time Garbage Collection	235
11.4	Link-Time Optimization	237
11.5	Link-Time Code Generation	238
11.6	The Java Linking Model	241
	Exercises	244
	Project	245

REFERENCES 246

INDEX 249

PREFACE

Linkers and loaders have been part of the software toolkit for almost as long as there have been computers. They are the critical tools that permit programs to be built from modules rather than as one big monolith.

As early as 1947, programmers started to use primitive loaders that could take program routines stored on separate tapes and combine and relocate them into one program. By the early 1960s, these loaders had evolved into full-fledged linkage editors. Because program memory remained expensive and limited and computers were (by modern standards) slow, these linkers contained complex features for creating complex memory overlay structures to squeeze large programs into small memory and for re-editing previously linked programs to save the time needed to rebuild a program from scratch.

During the 1970s and 1980s there was little progress in linking technology. Linkers tended to become even simpler, as virtual memory moved much of the job of storage management away from applications and overlays into the operating system; and as computers became faster and disks larger, it became easier to re-create a linked program from scratch to replace a few modules rather than to relink just the changes. In the 1990s linkers have again become more complex, adding support for modern features including dynamically linked shared libraries and the unusual demands of C++. Radical new processor architectures with wide instruction words and compiler-scheduled memory

accesses (such as the Intel IA64) will also put new demands on linkers to ensure that the complex requirements of the code are met in linked programs.

WHO IS THIS BOOK FOR?

This book is intended for several overlapping audiences: students, practicing programmers, and computer language designers and developers. Courses in compiler construction and operating systems have generally given scant treatment to linking and loading, often because the linking process seemed trivial or obvious. Although this was arguably true when the languages of interest were Fortran, Pascal, and C, and when operating systems didn't use memory mapping or shared libraries, it's much less true now. C++, Java, and other object-oriented languages require a much more sophisticated linking environment. Memory-mapped executable programs, shared libraries, and dynamic linking affect many parts of an operating system, and an operating system designer disregards linking issues at his or her peril.

Practicing programmers need to be aware of what linkers do, again particularly for modern languages. C++ places unique demands on a linker, and large C++ programs are prone to develop hard-to-diagnose bugs as a result of unexpected occurrences at link time. (The best known of these are static constructors that run in an order the programmer wasn't expecting.) Linker features such as shared libraries and dynamic linking offer great flexibility and power, when used appropriately.

Language designers and developers need to be aware of what linkers do and can do as they build languages and compilers. Programming tasks that had been handled by hand for 30 years are automated in C++, which depends on the linker to handle the details. (Consider what a programmer has to do to get the equivalent of C++ templates in C or to ensure that the initialization routines in each of a hundred C source files are called before the body of the program starts.) Future languages will automate even more program-wide bookkeeping tasks, with more powerful linkers doing the work. Linkers will also be more involved in global program optimization, because the linker is the only stage of the compiler process that handles the entire program's code together and thus can do transformations that affect the entire program as a unit.

People who write linkers also need this book, of course. But all the linker writers in the world could probably fit in one room and half of them already have copies because they reviewed the manuscript.

OVERVIEW

The key points for each chapter are highlighted with dots in the margin like the ones highlighting this paragraph. If you're inclined to skip over a chapter or two, please at least look at the highlighted points on your way through.

Chapter 1 provides a short historical overview of the linking process and discusses the stages of the linking process. It ends with a short but complete example of a linker run, from input object files to a runnable "Hello, world" program.

Chapter 2 reviews computer architecture from the point of view of linker design. It examines SPARC, a representative reduced instruction set architecture; the IBM 360/370, an old but still very viable register-memory architecture; and the Intel x86, which is in a category of its own. Important architectural aspects include memory architecture, program addressing architecture, and the layout of address fields in individual instructions.

Chapter 3 examines the internal structure of object and executable files. It starts with the very simplest files, DOS COM files, and goes on to examine progressively more complex files including DOS EXE, UNIX a.out and ELF, MS Windows COFF and PE (EXE and DLL), OS/360 object files, and Intel/Microsoft OMF.

Chapter 4 covers the first stage of linking—allocating storage to the segments of the linked program—with examples from real linkers.

Chapter 5 covers symbol binding and resolution, the process by which a symbolic reference in one file to a name in a second file is resolved to a machine address.

Chapter 6 covers the creation and use of object code libraries, along with issues of library structure and performance.

Chapter 7 covers address relocation, which is the process of adjusting the object code in a program to reflect the actual addresses at which it runs. It also covers position-independent code (PIC)—code created in a way that avoids the need for relocation—and the costs and benefits of doing so.

Chapter 8 covers the loading process, which involves getting a program from a file into the computer's memory in order to run it. It also covers tree-structured overlays, an old but still effective technique to conserve address space.

Chapter 9 looks at what's required to share a single copy of a library's code among many different programs. This chapter concentrates on statically linked shared libraries.

Chapter 10 broadens the discussion of Chapter 9 to dynamically linked shared libraries. It treats two examples in detail, Win32 dynamic-link libraries (DLLs) and UNIX/Linux ELF shared libraries.

Chapter 11 looks at a variety of tasks that sophisticated modern linkers do. It covers new features that C++ requires, including name mangling, global constructors and destructors, template expansion, and duplicate code elimination. Other techniques include incremental linking, link-time garbage collection, link-time code generation and optimization, load-time code generation, and profiling and instrumentation. The chapter concludes with an overview of the Java linking model, which is considerably more semantically complex than any of the other linkers covered.



THE PROJECT

Chapters 3 through 11 have a continuing project to develop a small but functional linker in perl. Although perl is an unlikely implementation language for a production linker, it's an excellent choice for a term project. Perl handles many of the low-level programming chores that bog down programming in languages like C or C++, letting the student concentrate on the algorithms and data structures of the project at hand. Perl is available at no charge on most current computers, including Windows 95/98 and NT, UNIX, and Linux, and many excellent books are available to teach perl to new users. (See the books by Hoffman, Schwartz, and Wall et al. listed in the references at the back of this book.)

The initial project in Chapter 3 builds a linker skeleton that can read and write files in a simple but complete object format, and subsequent chapters add functions to the linker until the final result is a full-fledged linker that supports shared libraries and produces dynamically linkable objects. Perl is quite able to handle arbitrary binary files and data structures, and the project linker could be adapted if desired to handle native object formats.

Object files for the project can be found at the supporting Web site for this book at *linker.iecc.com*.

ACKNOWLEDGMENTS

Many, many people generously contributed their time to reading and reviewing the manuscript of this book, including both the publisher's reviewers and

the readers of the *comp.compilers* usenet newsgroup who read and commented on an on-line version of the manuscript. In alphabetical order, they are Mike Albaugh, Rod Bates, Gunnar Blomberg, Robert Bowdidge, Keith Breinholt, Brad Brisco, Andreas Buschmann, David S. Cargo, John Carr, David Chase, Ben Combee, Ralph Corderoy, Paul Curtis, Bjorn De Sutter, Lars Duening, Phil Edwards, Oisin Feeley, Mary Fernandez, Michael Lee Finney, Peter H. Froehlich, Robert Goldberg, James Grosbach, Rohit Grover, Quinn Tyler Jackson, Colin Jensen, Glenn Kasten, Louis Krupp, Terry Lambert, Doug Landauer, Jim Larus, Len Lattanzi, Greg Lindahl, Peter Ludemann, Steven D. Majewski, John McEnerney, Larry Meadows, Jason Merrill, Carl Montgomery, Cyril Muerillon, Sameer Nanajkar, Jacob Navia, Simon Peyton-Jones, Allan Porterfield, Charles Randall, Thomas David Rivers, Ken Rose, Alex Rosenberg, Raymond Roth, Timur Safin, Kenneth G. Salter, Donn Seeley, Aaron F. Stanton, Harlan Stenn, Mark Stone, Robert Strandh, Ian Taylor, Michael Trofimov, Hans Walheim, and Roger Wong.

These people are responsible for most of the true statements in this book. The false ones remain the author's responsibility. (If you find any of the latter, please contact me at the address below so they can be fixed in subsequent printings.)

I particularly thank my editor Tim Cox and editorial assistant Sarah Luger at Morgan Kaufmann for putting up with my interminable delays during the writing process and for pulling all the pieces of this book together.

CONTACT US

This book has a supporting Web site at *linker.iecc.com*. It includes example chapters from the book, samples of perl code and object files for the project, and updates and errata.

You can send email to the author at *linker@iecc.com*. The author reads all the mail, but because of the volume received may not be able to answer all questions promptly.

LINKING AND LOADING

1.1 WHAT DO LINKERS AND LOADERS DO?

The basic job of any linker or loader is simple: It binds more abstract names to more concrete names, which permits programmers to write code using the more abstract names. For example, it takes a name written by a programmer such as `getLine` and binds it to “the location 612 bytes from the beginning of the executable code in module `iosys`.” Or it may take a more abstract numeric address such as “the location 450 bytes beyond the beginning of the static data for this module” and bind it to a numeric address.

1.2 ADDRESS BINDING: A HISTORICAL PERSPECTIVE

A useful way to get some insight into what linkers and loaders do is to look at their part in the development of computer programming systems. The earliest computers were programmed entirely in machine language. Programmers

would write out the symbolic programs on sheets of paper, hand-assemble them into machine code, and then toggle the machine code into the computer, or perhaps punch it on paper tape or cards. (Real hotshots could compose code directly at the switches.) If the programmer used symbolic addresses at all, the symbols were bound to addresses as the programmer did his or her hand translation. If it turned out that an instruction had to be added or deleted, the entire program had to be inspected and any addresses affected by the added or deleted instruction adjusted by hand.

The problem was that the names were bound to addresses too early in the process. Assemblers solved that problem by letting programmers write programs in terms of symbolic names, with the assembler binding the names to machine addresses. If the program changed, the programmer had to reassemble it, but the work of assigning the addresses is pushed off from the programmer to the computer.

Libraries of code compound the address assignment problem. Since the basic operations that computers can perform are so simple, useful programs are composed of subprograms that perform higher-level and more complex operations. Computer installations keep a library of prewritten and debugged subprograms that programmers can draw upon for use in new programs they write, rather than requiring programmers to write all their own subprograms. The programmer then loads the subprograms in with the main program to form a complete working program.

Programmers were using libraries of subprograms even before they used assemblers. By 1947, John Mauchly (leader of the ENIAC project) wrote about loading programs along with subprograms selected from a catalog of programs stored on tapes and of the need to relocate the subprograms' code to reflect the addresses at which they were loaded. Perhaps surprisingly, these two basic linker functions—relocation and library search—appear to predate even assemblers, as Mauchly expected both the program and subprograms to be written in machine language. The relocating loader allowed the authors and users of the subprograms to write each subprogram as though it would start at location zero and to defer the actual address binding until the subprograms were linked with a particular main program.

With the advent of operating systems, relocating loaders separate from linkers and libraries became necessary. Before there were operating systems, each program had the machine's entire memory at its disposal, so the

program could be assembled and linked for fixed memory addresses, based on the certainty that all addresses in the computer would be available. But with operating systems, the program had to share the computer's memory with the operating system and perhaps even with other programs. This meant that the actual addresses at which the program would be running weren't known until the operating system loaded the program into memory, with final address binding deferred past link time to load time. Linkers and loaders now divided up the work, with linkers doing part of the address binding, assigning relative addresses within each program, and the loader doing a final relocation step to assign actual addresses.

As systems became more complex, they called upon linkers to do more and more complex name management and address binding. Fortran programs used multiple subprograms and common blocks—areas of data shared by multiple subprograms—and it was up to the linker to lay out storage and to assign the addresses both for the subprograms and the common blocks. Linkers increasingly had to deal with object code libraries, including both application libraries written in Fortran and other languages and compiler support libraries that were called implicitly from compiled code to handle I/O and other high-level operations.

Programs quickly became larger than available memory, so linkers provided overlays, a technique that let programmers arrange for different parts of a program to share the same memory, with each overlay loaded on demand when another part of the program called into it. Overlays were widely used on mainframes from the advent of disks around 1960 until the spread of virtual memory in the mid-1970s; they reappeared on microcomputers in the early 1980s in exactly the same form and faded as virtual memory appeared on PCs in the 1990s. Overlays are still used in memory-limited embedded environments, and they may yet reappear in other places where precise programmer or compiler control of memory usage improves performance.

With the advent of hardware relocation and virtual memory, linkers and loaders actually became less complex, because each program could again have an entire address space. Programs could be linked to be loaded at fixed addresses, with hardware rather than software relocation taking care of any load-time relocation. However, computers with hardware relocation invariably run more than one program; frequently they run multiple copies of the same program. When a computer runs multiple copies of one program, some

parts of the program—the executable code, in particular—are the same among all running copies, while other parts are unique to each copy. If the parts that don't change can be separated out from the parts that do change, the operating system can use a single copy of the unchanging part, saving considerable storage. Compilers and assemblers were modified to create object code in multiple sections, with one section for read-only code and another section for writable data. The linker had to be able to combine all of the sections of each type so that the linked program would have all the code in one place and all the data in another. This didn't delay address binding any more than before, because addresses were still assigned at link time, but more work was deferred to the linker to assign addresses for all the sections.

Even when different programs are running on a computer, those different programs usually turn out to share a lot of common code. For example, nearly every program written in C uses routines such as `fopen` and `printf`; database applications all use a large access library to connect to the database; and programs running under a graphical user interface (GUI) such as X Windows, MS Windows, or the Macintosh all use pieces of the GUI library. Most systems now provide shared libraries for programs to use, so that all the programs that use a library can share a single copy of it. This both improves run-time performance and saves a lot of disk space; in small programs the common library routines often take up more space than the program itself.

In the simpler static shared libraries, each library is bound to specific addresses at the time the library is built, and the linker binds program references to library routines to those specific addresses at link time. Static libraries turn out to be inconveniently inflexible, because programs potentially have to be relinked every time any part of the library changes, and the details of creating static shared libraries turn out to be very tedious. Systems therefore added dynamically linked libraries in which library sections and symbols aren't bound to actual addresses until the program that uses the library starts running. Sometimes the binding is delayed even further than that; with full-fledged dynamic linking, the addresses of called procedures aren't bound until the first call. Furthermore, programs can bind to libraries as the programs are running, loading libraries in the middle of program execution. This provides a powerful and high-performance way to extend the function of programs. Microsoft Windows in particular makes extensive use of run-time loading of shared libraries—called dynamically linked libraries (DLLs)—to construct and extend programs.

1.3 LINKING VS. LOADING

Linkers and loaders perform several related but conceptually separate actions.

- *Program loading.* They copy a program from secondary storage (which, since about 1968, invariably means a disk) into main memory so it's ready to run. In some cases loading just involves copying the data from disk to memory; in others it involves allocating storage, setting protection bits, or arranging for virtual memory to map virtual addresses to disk pages.
- *Relocation.* Compilers and assemblers generally create each file of object code with the program addresses starting at zero, but few computers let you load your program at location zero. If a program is created from multiple subprograms, all the subprograms have to be loaded at non-overlapping addresses. Relocation is the process of assigning load addresses to the various parts of the program and adjusting the code and data in the program to reflect the assigned addresses. In many systems, relocation happens more than once. It's quite common for a linker to create a program from multiple subprograms and to create one linked output program that starts at zero, with the various subprograms relocated to locations within the big program. Then when the program is loaded, the system picks the actual load address and the linked program is relocated as a whole to the load address.
- *Symbol resolution.* When a program is built from multiple subprograms, the references from one subprogram to another are made using symbols; a main program might use a square root routine called `sqrt`, and the math library defines `sqrt`. A linker resolves the symbol by noting the location assigned to `sqrt` in the library and patching the caller's object code so the call instruction refers to that location.

Although there's considerable overlap between linking and loading, it's reasonable to define a program that does program loading as a loader, and one that does symbol resolution as a linker. Either can do relocation, and there have been all-in-one linking loaders that do all three functions.

The line between relocation and symbol resolution can be fuzzy. Because linkers already can resolve references to symbols, one way to handle code relocation is to assign a symbol to the base address of each part of the program and to treat relocatable addresses as references to the base address symbols.

One important feature that linkers and loaders share is that they both patch object code; as such they are the only widely used programs to do so other than perhaps debuggers. This is a uniquely powerful feature, albeit one that is extremely machine specific in the details and that can lead to baffling bugs if done wrong.

TWO-PASS LINKING

Now we turn to the general structure of linkers. Linking, like compiling or assembling, is fundamentally a two-pass process. A linker takes as its input a set of input object files, libraries, and perhaps command files, and produces as its result an output object file and perhaps ancillary information such as a load map or a file containing debugger symbols (Figure 1.1).

Each input file contains a set of *segments*, contiguous blocks of code or data to be placed in the output file. Each input file also contains at least one symbol table. Some symbols are exported—defined within the file for use in other files, generally the names of routines within the file that can be called from elsewhere. Other symbols are imported—used in the file but not defined, generally the names of routines called from but not present in the file.

When a linker runs, it first has to scan the input files to find the sizes of the segments and to collect the definitions and references of all of the symbols. It creates a segment table listing all of the segments defined in the input files and a symbol table with all of the symbols imported or exported. Using the data from the first pass, the linker assigns numeric locations to symbols, determines the sizes and location of the segments in the output address space, and figures out where everything goes in the output file.

The second pass uses the information collected in the first pass to control the actual linking process. It reads and relocates the object code, substituting numeric addresses for symbol references and adjusting memory addresses in code and data to reflect relocated segment addresses, and writes the relocated code to the output file. It then writes the output file, generally with header information, the relocated segments, and symbol table information. If the

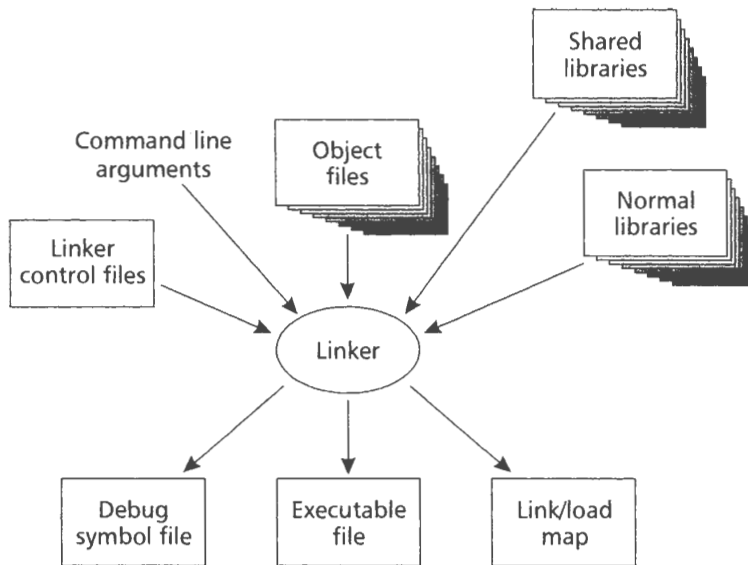


FIGURE 1.1 • The linker process.

program uses dynamic linking, the symbol table contains the information the run-time linker will need to resolve dynamic symbols. In many cases, the linker itself will generate small amounts of code or data in the output file, such as glue code used to call routines in overlays or in dynamically linked libraries or an array of pointers to initialization routines that need to be called at program startup time. Whether or not the program uses dynamic linking, the file may also contain a symbol table for relinking or debugging that isn't used by the program itself but that may be used by other programs that deal with the output file.

Some object formats are relinkable; that is, the output file from one linker run can be used as the input to a subsequent linker run. This requires that the output file contain a symbol table like that in an input file, as well as all of the other auxiliary information present in an input file.

Nearly all object formats have provision for debugging symbols, so that when the program is run under the control of a debugger, the debugger can use those symbols to let the programmer control the program in terms of the line numbers and names used in the source program. Depending on the

details of the object format, the debugging symbols may be intermixed in a single symbol table with symbols needed by the linker, or there may be one table for the linker and a separate, somewhat redundant table for the debugger.

A few linkers appear to work in one pass. They do so by first buffering some or all of the contents of the input file in memory or disk during the linking process, then reading the buffered material later. Since this is an implementation trick that doesn't fundamentally affect the two-pass nature of linking, we don't address it further here.

OBJECT CODE LIBRARIES

All linkers support object code libraries in one form or another, with most also providing support for various kinds of shared libraries. The basic principle of an object code library is simple enough (Figure 1.2). A library is little more than a set of object code files. (Indeed, on some systems you can literally catenate a group of object files together and use the result as a link library.) If any imported names remain undefined after the linker processes all of the regular input files, it runs through the library or libraries and links in any of the files in the library that export one or more undefined names.

Shared libraries complicate this task a little by moving some of the work from link time to load time. The linker identifies the shared libraries that resolve the undefined names in a linker run, but rather than linking anything into the program, the linker notes in the output file the names of the libraries in which the symbols were found, so that the shared library can be bound in when the program is loaded. See Chapters 9 and 10 for the details.

RELOCATION AND CODE MODIFICATION

The heart of a linker's or loader's actions is relocation and code modification. When a compiler or assembler generates an object file, it generates the code using the unrelocated addresses of code and data defined within the file and usually zeros for code and data defined elsewhere. As part of the linking process, the linker modifies the object code to reflect the actual addresses assigned. For example, consider the following piece of x86 code that moves the contents of variable *a* to variable *b* using the *eax* register:

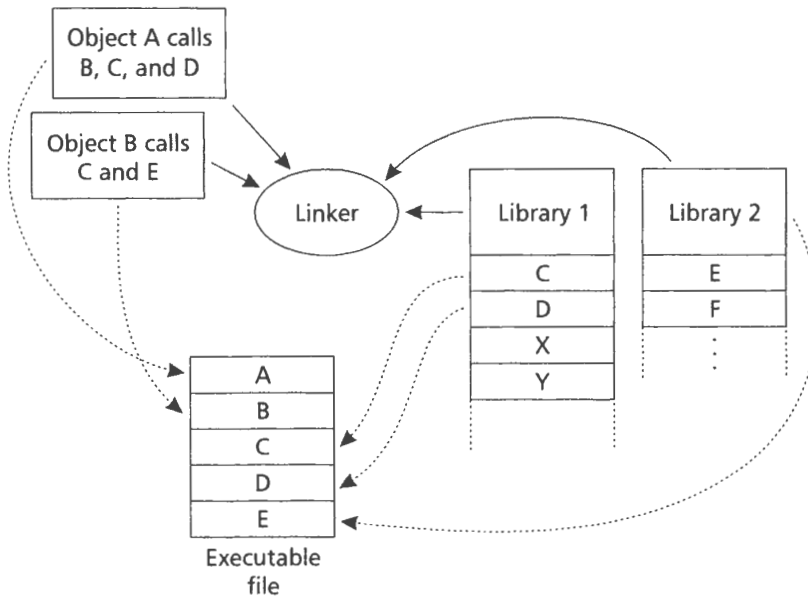


FIGURE 1.2 • Object code libraries.

```

mov a,%eax
mov %eax,b

```

If *a* is defined in the same file at location hex 1234 and *b* is imported from somewhere else, the generated object code will be

```

A1 34 12 00 00 mov a,%eax
A3 00 00 00 00 mov %eax,b

```

Each instruction contains a 1-byte operation code followed by a 4-byte address. The first instruction has a reference to 1234 (byte-reversed, because the x86 uses a right-to-left byte order) and the second instruction has a reference to zero because the location of *b* is unknown.

Now assume that the linker links this code so that the section in which *a* is located is relocated by hex 10000 bytes, and assume that *b* turns out to be at hex 9A12. The linker modifies the code to be

```

A1 34 12 01 00 mov a,%eax
A3 12 9A 00 00 mov %eax,b

```

That is, the linker adds 10000 to the address in the first instruction so that now the instruction refers to a's relocated address (11234) and the linker patches in the address for b. These adjustments affect instructions, but any pointers in the data part of an object file have to be adjusted as well.

On older computers with small address spaces and direct addressing, the modification process is fairly simple, because there are only one or two address formats that a linker has to handle. Modern computers, including all RISCs, require considerably more complex code modification. No single instruction contains enough bits to hold a direct address, so the compiler and linker have to use complicated addressing tricks to handle data at arbitrary addresses. In some cases, it's possible to concoct an address using two or three instructions, each of which contains part of the address, and to use bit manipulation to combine the parts into a full address. In such a case, the linker has to be prepared to modify each of the instructions appropriately, inserting some of the bits of the address into each instruction. In other cases, all of the addresses used by a routine or group of routines are placed in an array that is used as an address pool—initialization code sets one of the machine registers to point to that array, and code loads pointers out of the address pool as needed using that register as a base register. The linker may have to create the array from all of the addresses used in a program and then modify instructions that so that they refer to the appropriate address pool entry. We discuss this in Chapter 7.

Some systems require position-independent code that will work correctly regardless of where in the address space it is loaded. Linkers generally have to provide extra tricks to support that, separating out the parts of the program that can't be made position independent and arranging for the two parts to communicate. (See Chapter 8.)

1.4 COMPILER DRIVERS

In most cases, the operation of the linker is invisible to the programmer (or nearly so) because it's run automatically as part of the compilation process. Most compilation systems have a compiler driver that automatically invokes

the phases of the compiler as needed. For example, if the programmer has two C language source files, the compiler driver will run a sequence of programs like this on a UNIX system:

1. C preprocessor on file A, creating preprocessed A
2. C compiler on preprocessed A, creating assembler file A
3. Assembler on assembler file A, creating object file A
4. C preprocessor on file B, creating preprocessed B
5. C compiler on preprocessed B, creating assembler file B
6. Assembler on assembler file B, creating object file B
7. Linker on object files A and B and system C library

That is, it compiles each source file to assembler and then object code and links the object code together, including any needed routines from the system C library.

Compiler drivers are often much more clever than this. Many will compare the creation dates of source and object files and only recompile source files that have changed. (The UNIX `make` program is the classic example.) Particularly when compiling C++ and other object-oriented languages, compiler drivers can play all sorts of tricks to work around limitations in linkers or object formats. For example, C++ templates define a potentially infinite set of related routines, so to find the finite set of template routines that a program actually uses, a compiler driver can link the program's object files together with no template code, read the error messages from the linker to see what's undefined, call the C++ compiler to generate object code for the necessary template routines, and relink. We cover some of these tricks in Chapter 11.

LINKER COMMAND LANGUAGES

Every linker has some sort of command language to control the linking process. At the very least the linker needs the list of object files and libraries to link. Generally there is a long list of possible options: whether to keep debugging symbols, whether to use shared or unshared libraries, which of several possible output formats to use. Most linkers permit some way to specify the address at which the linked code is to be bound, which comes in handy when

we are using a linker to link a system kernel or other program that doesn't run under control of an operating system. In linkers that support multiple code and data segments, a linker command language can specify the order in which segments are to be linked, special treatment for certain kinds of segments, and other application-specific options.

There are four common techniques to pass commands to a linker:

1. *Command line.* Most systems have a command line or the equivalent, by which we can pass a mixture of file names and switches. This is the usual approach for UNIX and MS Windows linkers. On systems with limited-length command lines, there's usually a way to direct the linker to read commands from a file and to treat them as though they were on the command line.
2. *Intermixed with object files.* Some linkers, such as IBM mainframe linkers, accept alternating object files and linker commands in a single input file. This dates from the era of card decks, when the programmer would pile up object decks and hand-punched command cards in a card reader.
3. *Embedded in object files.* Some object formats, notably Microsoft's, permit linker commands to be embedded inside object files. This permits a compiler to pass any options needed to link an object file in the file itself. For example, the C compiler passes commands to search the standard C library.
4. *Separate configuration language.* A few linkers have a full-fledged configuration language to control linking. The GNU linker, which can handle an enormous range of object file formats, machine architectures, and address space conventions, has a complex control language that lets a programmer specify the order in which segments should be linked, rules for combining similar segments, segment addresses, and a wide range of other options. Other linkers have less complex languages to handle specific features such as programmer-defined overlays.

1.5 LINKING: A TRUE-LIFE EXAMPLE

We complete our introduction to linking with a small but real linking example. Figure 1.3 shows a pair of C language source files, `m.c` with a main program

Source file `m.c`

```
extern void a(char *);
int main(int ac, char **av)
{
    static char string[] = "Hello, world!\n";
    a(string);
}
```

Source file `a.c`

```
#include <unistd.h>
#include <string.h>
void a(char *s)
{
    write(1, s, strlen(s));
}
```

FIGURE 1.3 • Source files.

that calls a routine named `a` and `a.c` that contains the routine with a call to the library routines `strlen` and `write`.

On my Pentium with GCC, the main program `m.c` compiles into a 165-byte object file in the classic `a.out` object format (Figure 1.4). That object file includes a fixed-length header, 16 bytes of text segment containing the read-only program code, and 16 bytes of data segment containing the string. Following that are two relocation entries, one that marks the `pushl` instruction that puts the address of the string on the stack in preparation for the call to `a` and one that marks the `call` instruction that transfers control to `a`. The symbol table exports the definition of `_main`, imports `_a`, and contains a couple of other symbols for the debugger. (Each global symbol is prefixed with an underscore, for reasons described in Chapter 5.) Note that the `pushl` instruction refers to location hex 10, the tentative address for the string, because it's in the same object file, while the `call` refers to location 0 since the address of `_a` is unknown.

The subprogram file `a.c` compiles into a 160-byte object file (Figure 1.5), with the header, a 28-byte text segment, and no data. Two relocation entries mark the calls to `strlen` and `write`, and the symbol table exports `_a` and imports `_strlen` and `_write`.

```

Sections:
  Idx Name      Size      VMA      LMA      File off  Algn
    0 .text  00000010  00000000  00000000  00000020  2**3
    1 .data  00000010  00000010  00000010  00000030  2**3
Disassembly of section .text: 00000000 <_main>:
    0:    55                      pushl %ebp
    1:   89 e5                  movl %esp,%ebp
    3:   68 10 00 00 00        pushl $0x10
        4: 32      .data
    8:   e8 f3 ff ff ff        call 0
        9: DISP32 _a
    d:   c9                      leave
    e:   c3                      ret
    ...

```

FIGURE 1.4 • Object code for m.c.

To produce an executable program, the linker combines these two object files with a standard startup initialization routine for C programs and with necessary routines from the C library, as displayed (in part) in Figure 1.6.

The linker combines corresponding segments from each input file, so there is one combined text segment, one combined data segment, and one bss segment (zero-initialized data, which the two input files didn't use). Each segment is padded out to a 4KB boundary to match the x86 page size, so the text segment is 4KB (minus a 32-byte a.out header present in the file but not logically part of the segment); the data and bss segments are also each 4KB.

The combined text segment contains the text of library startup code called start-c, then text from m.o relocated to 10a4, text from a.o relocated to 10b4, and routines linked from the C library relocated to higher addresses in the text segment. The data segment, not displayed in Figure 1.6, contains the combined data segments in the same order as the text segments. Since the code for _main has been relocated to address hex 10a4, that address is patched into the call instruction in start-c. Within the main routine, the reference to the string is relocated to hex 2024, the string's final location in the data segment, and the call is patched to 10b4, the final address of _a. Within _a, the calls to _strlen and _write are patched to the final addresses for those two routines.

```

Sections:
Idx Name      Size      VMA      LMA      File off  Algn
  0 .text     0000001c  00000000  00000000  00000020  2**2
               CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data     00000000  0000001c  0000001c  0000003c  2**2
               CONTENTS, ALLOC, LOAD, DATA
Disassembly of section .text: 00000000 <_a>:
  0:   55                      pushl   %ebp
  1:   89 e5                   movl    %esp,%ebp
  3:   53                      pushl   %ebx
  4:   8b 5d 08                movl    0x8(%ebp),%ebx
  7:   53                      pushl   %ebx
  8:   e8 f3 ff ff ff          call    0
                        9: DISP32    _strlen
 d:   50                      pushl   %eax
 e:   53                      pushl   %ebx
 f:   6a 01                  pushl   $0x1
11:   e8 ea ff ff ff          call    0
                        12: DISP32    _write
16:   8d 65 fc                leal    -4(%ebp),%esp
19:   5b                      popl    %ebx
1a:   c9                      leave
1b:   c3                      ret

```

FIGURE 1.5 • Object code for a.c.

The executable program also contains about a dozen other routines from the C library (also not shown) that are called directly or indirectly from the startup code or from `_write` (error routines, in the latter case). The executable contains no relocation data, because this file format is not relinkable and the operating system loads it at a known fixed address. It contains a symbol table for the benefit of a debugger, although the executable doesn't use the symbols and the symbol table can be stripped off to save space.

In this example, the code linked from the library is considerably larger than the code for the program itself. That's quite common, particularly when programs use large graphics or windowing libraries; this provides the impetus

16 Chapter 1 • Linking and Loading

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000fe0	00001020	00001020	00000020	2**3
1	.data	00001000	00002000	00002000	00001000	2**3
2	.bss	00000000	00003000	00003000	00000000	2**3

Disassembly of section .text: 00001020 <start-c>:

```

...
1092:    e8 0d 00 00 00    call 10a4 <_main>
...
000010a4 <_main>:
10a4: 55                pushl %ebp
10a5: 89 e5            movl %esp,%ebp
10a7: 68 24 20 00 00    pushl $0x2024
10ac: e8 03 00 00 00    call 10b4 <_a>
10b1: c9              leave
10b2: c3              ret
000010b4 <_a>:
10b4: 55                pushl %ebp
10b5: 89 e5            movl %esp,%ebp
10b7: 53              pushl %ebx
10b8: 8b 5d 08        movl 0x8(%ebp),%ebx
10bb: 53              pushl %ebx
10bc: e8 37 00 00 00    call 10f8 <_strlen>
10c1: 50              pushl %eax
10c2: 53              pushl %ebx
10c3: 6a 01          pushl $0x1
10c5: e8 a2 00 00 00    call 116c <_write>
10ca: 8d 65 fc        leal -4(%ebp),%esp
10cd: 5b              popl %ebx
10ce: c9              leave
10cf: c3              ret
...
000010f8 <_strlen>:
...
0000116c <_write>:
...
```

FIGURE 1.6 • Selected parts of executable program.

for shared libraries, as discussed in Chapters 9 and 10. The linked program is 8KB, but the identical program linked using shared libraries is only 264 bytes. This is a toy example, of course, but real programs often have equally dramatic space savings.

EXERCISES

EXERCISE • 1.1

What is the advantage of separating linker and loader into separate programs? Under what circumstances would a combined linking loader be useful?

EXERCISE • 1.2

Nearly every programming system produced in the past 50 years includes a linker. Why?

EXERCISE • 1.3

In this chapter we've discussed linking and loading assembled or compiled machine code. Would a linker or loader be useful in a purely interpretive system that directly interprets source language code? How about in an interpretive system that turns the source into an intermediate representation like P-code or the Java Virtual Machine?

ARCHITECTURAL ISSUES

Linkers and loaders—along with compilers and assemblers—are exquisitely sensitive to the architectural details, both the hardware architecture and the architecture conventions required by the operating system of their target computers. In this chapter we cover enough computer architecture to understand the jobs that linkers have to do. The descriptions of the computer architectures in this chapter are deliberately incomplete and leave out the parts that don't affect the linker (such as floating point and I/O).

Two aspects of hardware architecture affect linkers: program addressing and instruction formats. One of the things that a linker does is to modify addresses and offsets both in data memory and in instructions. In both cases, the linker has to ensure that its modifications match the addressing scheme that the computer uses; when modifying instructions it must further ensure that the modifications don't result in an invalid instruction. At the end of the chapter, we also look at address space architecture, that is, what set of addresses a program has to work with.

2.1 APPLICATION BINARY INTERFACES

Every operating system presents an *application binary interface* (ABI) to programs that run under that system. The ABI consists of programming conventions that applications have to follow to run under the operating system. ABIs invariably include a set of system calls and the technique to invoke the system calls, as well as rules about what memory addresses a program can use and often rules about usage of machine registers. From the point of view of an application, the ABI is as much a part of the system architecture as the underlying hardware architecture, because a program will fail equally badly if it violates the constraints of either.

In many cases, the linker has to do a significant part of the work involved in complying with the ABI. For example, if the ABI requires that each program must contain a table of all of the addresses of static data used by routines in the program, the linker often creates that table, by collecting address information from all of the modules linked into the program. The aspect of the ABI that most often affects the linker is the definition of a standard procedure call, a topic we return to later in this chapter.

2.2 MEMORY ADDRESSES

Every computer includes a main memory. The main memory is an array of storage locations, with each location having a numeric address. The addresses start at zero and run up to some large number determined by the number of bits in an address.

BYTE ORDER AND ALIGNMENT

Each storage location consists of a fixed number of bits. Over the past 50 years computers have been designed with storage locations consisting of as many as 64 bits and as few as 1 bit, but now nearly every computer in production addresses 8-bit bytes. Since many of the data that computers handle, notably program addresses, are bigger than 8 bits, the computers can also handle 16-,

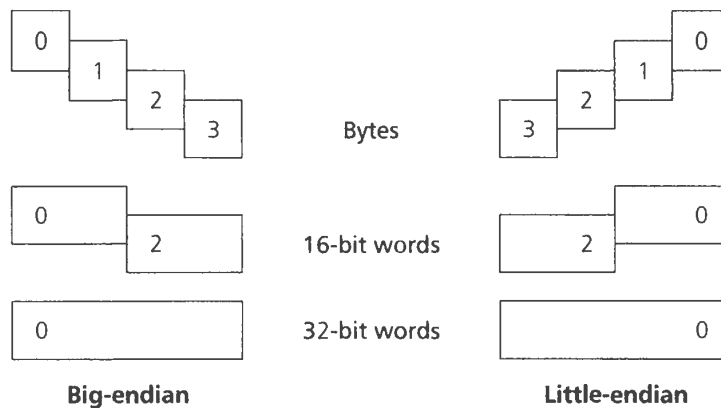


FIGURE 2.1 • Byte-addressable memory.

32-, and often 64- or 128-bit data as well, with multiple adjacent bytes grouped together. On some computers, notably those from IBM and Motorola, the first (numerically lowest-addressed) byte in multibyte data is the most significant byte, while others, notably DEC and Intel, it's the least significant byte (Figure 2.1). In a nod to *Gulliver's Travels*, the IBM/Motorola byte order scheme is known as "big-endian" while the DEC/Intel scheme is called "little-endian."

The relative merits of the two schemes have provoked vehement arguments over the years. In practice the major issue determining the choice of byte order is compatibility with older systems, since it is considerably easier to port programs and data between two machines with the same byte order than between machines with different byte orders. Many recent chip designs can support either byte order, with the choice made either by the way the chip is wired up, by programming at system boot time, or in a few cases it is even selected per application. (On these switch-hitting chips, the byte order of data handled by load and store instructions changes, but the byte order of constants encoded in instructions doesn't. This is the sort of detail that keeps the life of the linker writer interesting.)

Multibyte data must usually be aligned on a natural boundary. That is, 4-byte data should be aligned on a 4-byte boundary, 2-byte on 2-byte, and so forth. Another way to think of it is that the address of any N -byte datum should

have at least $\log_2 N$ low zero bits. On some systems (Intel x86, DEC VAX, IBM 370/390), misaligned data references work at the cost of reduced performance, while on others (most RISC chips) misaligned data cause a program fault. Even on systems where misaligned data don't cause a fault, the performance loss is usually great enough that it's worth the effort to maintain alignment where possible.

Many processors also have alignment requirements for program instructions. Most RISC chips require that instructions be aligned on 4-byte boundaries.

Each architecture also defines *registers*, a small set of fixed-length high-speed memory locations to which program instructions can refer directly. The number of registers varies from one architecture to another, from as few as eight in the Intel architecture to 32 in some RISC designs. Registers are almost invariably the same size as a program address; that is, on a system with 32-bit addresses, the registers are 32 bits, and on a system with 64-bit addresses, the registers are 64 bits as well.

2.3 ADDRESS FORMATION

As a computer program executes, it loads and stores data to and from memory, as determined by instructions in the program. The instructions are themselves stored in memory, usually a different part of memory from the program's data. Instructions are logically executed in the sequence they are stored, except that *jump* instructions specify a new place in the program to start executing instructions. (Some architectures use the term *branch* for some or all jumps, but we call them all jumps here.) Each instruction that references data memory and each jump specifies the address or addresses of the data to load or store or of the instruction to jump to. All computers have a variety of instruction formats and address formation rules that linkers have to be able to handle as they relocate addresses in instructions.

Although computer designers have come up with innumerable different and complex addressing schemes over the years, most computers currently in production have a relatively simple addressing scheme. (Designers found that it's hard to build a fast version of a complicated architecture, and compilers rarely make good use of complicated addressing features.) We'll use the following three architectures as examples:

- The IBM 360/370/390 (which we'll refer to as the 370). Although this is one of the oldest architectures still in use, its relatively clean design has worn well despite 35 years of added features and has been implemented in chips comparable in performance to modern RISCs.
- SPARC V8 and V9. This is a popular RISC architecture, with fairly simple addressing. V8 uses 32-bit registers and addresses; V9 adds 64-bit registers and addresses. The SPARC design is similar to other RISC architectures such as MIPS and Alpha.
- The Intel 386/486/Pentium (henceforth referred to as the x86). This is one of the most arcane and irregular architectures still in use, but it is undeniably the most popular.

2.4 INSTRUCTION FORMATS

Each architecture has several different instruction formats. We'll only address the format details relative to program and data addressing, because those are the main details that affect the linker. The 370 uses the same format for data references and jumps, while SPARC has different formats and the x86 has some common formats and some different.

Each instruction consists of an opcode, which determines what the instruction does, and operands. An operand may be encoded in the instruction itself (an *immediate* operand) or located in memory. The address of each operand in memory has to be calculated somehow. Sometimes the address is contained in the instruction (*direct addressing*). More often the address is found in one of the registers (*register indirect*) or calculated by adding a constant in the instruction to the contents of a register. If the value in the register is the address of a storage area, and the constant in the instruction is the offset of the desired datum in the storage area, this scheme is known as *based addressing*. If the roles are swapped and the register contains the offset, the scheme is known as *indexed addressing*. The distinction between based and indexed addressing isn't well defined, and many architectures combine them (e.g., the 370 has an addressing mode that adds together two registers and a constant in the instruction, arbitrarily calling one of the registers the base register and the other the index register, although the two are treated the same). Other more complicated address calculation schemes are still in use, but for

the most part the linker doesn't have to worry about them because they don't contain any fields the linker has to adjust.

Some architectures use fixed-length instructions, and some use variable-length instructions. All SPARC instructions are 4 bytes long, aligned on 4-byte boundaries. IBM 370 instructions can be 2, 4, or 6 bytes long, with the first 2 bits of the first byte determining the length and format of the instruction. Intel x86 instructions can be anywhere from 1 byte to 14 bytes long. The encoding is quite complex, partly because the x86 was originally designed for limited-memory environments with a dense instruction encoding and partly because the new instructions added in the 286, 386, and later chips had to be shoe-horned into unused bit patterns in the existing instruction set. Fortunately, from the point of view of a linker writer, the address and offset fields that a linker has to adjust all occur on byte boundaries, so the linker generally need not be concerned with the instruction encoding.

2.5 PROCEDURE CALLS AND ADDRESSABILITY

In the earliest computers, memories were small, and each instruction contained an address field large enough to contain the address of any memory location in the computer, a scheme now called direct addressing. By the early 1960s, addressable memory was getting large enough that an instruction set with a full address in each instruction would have large instructions that took up too much of still-precious memory. To solve this problem, computer architects abandoned direct addressing in some or all of the memory reference instructions, using index and base registers to provide most or all of the bits used in addressing. This allowed instructions to be shorter, at the cost of more complicated programming.

On architectures without direct addressing—such as the IBM 370 and SPARC—programs have a bootstrapping problem for data addressing. A routine uses base values in registers to calculate data addresses, but the standard way to get a base value into a register is to load it from a memory location, which is in turn addressed from another base value in a register. The bootstrap problem is to get the first base value into a register at the beginning of the program and subsequently to ensure that each routine has the base values it needs to address the data it uses.

PROCEDURE CALLS

Every ABI defines a standard procedure call sequence, using a combination of hardware-defined call instructions and conventions about register and memory use. A hardware call instruction saves the return address (the address of the instruction after the call) and jumps to the procedure. On architectures with a hardware stack, such as the x86, the return address is pushed on the stack, while on other architectures it's saved in a register, with software having the responsibility to save the register in memory if necessary. Architectures with a stack generally have a hardware return instruction that pops the return address from the stack and jumps to that address, while other architectures use a "branch to address in register" instruction to return.

Within a procedure, data addressing falls into four categories:

1. The caller can pass *arguments* to the procedure.
2. *Local variables* are allocated within the procedure and freed before the procedure returns.
3. *Local static data* is stored in a fixed location in memory and is private to the procedure.
4. *Global static data* is stored in a fixed location in memory and can be referenced from many different procedures.

The piece of stack memory allocated for a single procedure call is known as a *stack frame*. Figure 2.2 shows a typical stack frame. Arguments and local variables are usually allocated on the stack. One of the registers serves as a stack pointer, which can be used as a base register. In a common variant of this scheme, used with SPARC and x86, a separate frame pointer or base pointer register is loaded from the stack pointer at the time a procedure starts. This makes it possible to push variable-sized objects on the stack, changing the value in the stack pointer register to a hard-to-predict value, but still lets the procedure address arguments and locals at fixed offsets from the frame pointer (which doesn't change during a procedure's execution). Assuming that the stack grows from higher to lower addresses and that the frame pointer points to the address in memory where the return address is stored, arguments are at small positive offsets from the frame pointer, and local variables are at negative offsets. The operating system usually sets the initial stack

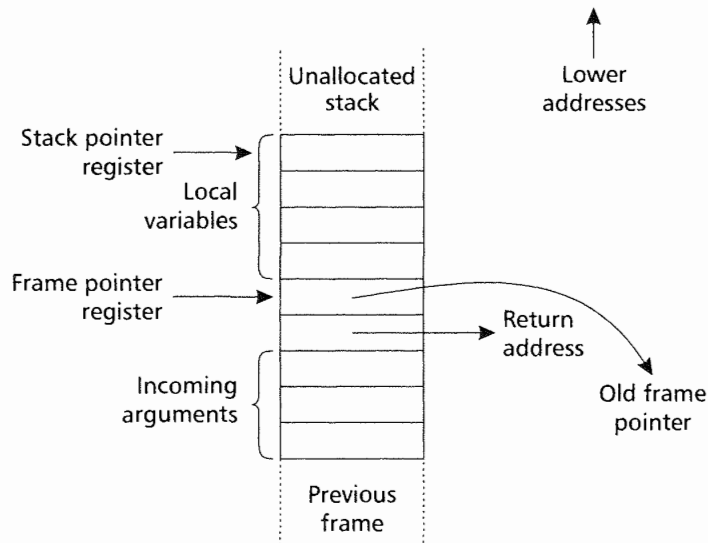


FIGURE 2.2 • Stack frame memory layout.

pointer register before a program starts, so the program need only update the register when it pushes and pops data.

For local and global static data, a compiler can generate a table of pointers to all of the static objects that a routine references. If one of the registers contains a pointer to this table, the routine can address any desired static object by loading the pointer to the object from the table using the table pointer register into another register, then using that second register as the base register to address the object. The trick, then, is to get the address of the table into the first register. On SPARC, the routine can load the table address into the register using a sequence of instructions with immediate operands, and on the 370 the routine can use a variant of a subroutine call instruction to load the program counter (the register that keeps the address of the current instruction) into a base register, although for reasons we discuss later, those techniques cause problems in library code. A better solution is to foist off the job of loading the table pointer on the routine's caller, because the caller will have its own table

```

... push arguments on the stack ...
store Rt -> xxx(Rf) ; save caller's table pointer in caller's stack frame
load Rx <- MMM(Rt)  ; load address of called routine into temp register
load Rt <- NNN(Rt)  ; load called routine's table pointer
call (Rx) ; call routine at address in Rx
load Rt <- xxx(Rf)  ; restore caller's table pointer

```

FIGURE 2.3 • Idealized calling sequence.

pointer already loaded and can get the address of the called routine's table from its own table.

Figure 2.3 shows a typical routine calling sequence. Rf is the frame pointer, Rt is the table pointer, and Rx is a temporary scratch register. The caller saves its own table pointer in its own stack frame, then loads both the address of the called routine and the called routine's pointer table into registers, then makes the call. The called routine can then find all of its necessary data using the table pointer in Rt, including addresses and table pointers for any routines that it in turn calls.

Several optimizations are often possible. In many cases, all of the routines in a module share a single pointer table, in which case intramodule calls needn't change the table pointer. The SPARC convention is that an entire library shares a single table—created by the linker—so the table pointer register can remain unchanged in intramodule calls. Calls within the same module can usually be made using a version of the `call` instruction with the offset to the called routine encoded in the instruction, which avoids the need to load the address of the routine into a register. With both of these optimizations, the calling sequence to a routine in the same module reduces to a single `call` instruction.

To return to the address bootstrap question, how does this chain of table pointers get started? If each routine gets its table pointer loaded by the preceding routine, where does the initial routine get its pointer? The answer varies, but it always involves special-case code. The main routine's table may be stored at a fixed address, or the initial pointer value may be tagged in the executable file so the operating system can load it before the program starts. No matter what the technique is, it invariably needs some help from the linker.

2.6 DATA AND INSTRUCTION REFERENCES

We now look more concretely at the way that programs in our three architectures address data values.

IBM 370

The 1960s-vintage System/360 started with a very straightforward data addressing scheme, which has become somewhat more complicated over the years as the 360 evolved into the 370 and 390. Every instruction that references data memory calculates the address by adding a 12-bit unsigned offset in the instruction to a base register and maybe to an index register. There are 16 general registers, each 32 bits, numbered from 0 to 15, all but one of which can be used as index registers. If register 0 is specified in an address calculation, the value 0 is used rather than the register contents. (Register 0 exists for arithmetic, but not for addressing.) In instructions that take the target address of a jump from a register, register 0 means don't jump.

Figure 2.4 shows the major instruction formats. An RX instruction contains a register operand and a single memory operand, whose address is calculated by adding the offset in the instruction to a base register and an index register. More often than not, the index register is zero so the address is just base plus offset. In the RS, SI, and SS formats, the 12-bit offset is added to a base register. An RS instruction has one memory operand, with one or two other operands being in registers. An SI instruction has one memory operand, the other operand being an immediate 8-bit value in the instruction. An SS instruction has two memory operands, for storage-to-storage operations. The RR format has two register operands and no memory operands at all, although some RR instructions interpret one or both of the registers as pointers to memory. The 370 and 390 added some minor variations on these formats, but none with different data addressing formats.

Instructions can directly address the lowest 4096 locations in memory by specifying base register zero. This ability is essential in low-level system programming but is never used in application programs, all of which use base register addressing.

Note that in all three instruction formats, the 12-bit address offset is always stored as the low 12 bits of a 16-bit aligned halfword. This makes it

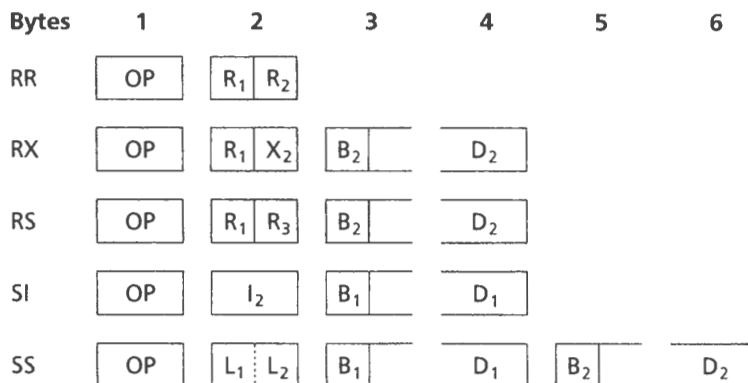


FIGURE 2.4 • IBM 370 instruction formats.

possible to specify fixups to address offsets in object files without any reference to instruction formats, because the offset format is always the same.

The original 360 had 24-bit addressing, with an address in memory or a register being stored in the low 24 bits of a 32-bit word and the high 8 bits disregarded. The 370 extended addressing to 31 bits. Unfortunately, many programs—including OS/360, the most popular operating system—stored flags or other data in the high byte of 32-bit address words in memory, so it wasn't possible to extend the addressing to 32 bits in the obvious way and still support existing object code. Instead, the system has 24-bit and 31-bit modes, and at any moment a CPU interprets 24-bit addresses or 31-bit addresses. A convention enforced by a combination of hardware and software states that an address word with the high bit set contains a 31-bit address in the rest of the word, while one with the high bit clear contains a 24-bit address. As a result, a linker has to be able to handle both 24-bit and 31-bit addresses because programs can and do switch modes depending on how long ago a particular routine was written. For historical reasons, 370 linkers also handle 16-bit addresses, because early small models in the 360 line often had 64KB or less of main memory and programs used load and store halfword instructions to manipulate address values.

Later models of the 370 and 390 added segmented address spaces somewhat like those of the x86 series. These features let the operating system define multiple 31-bit address spaces that a program can address, with extremely complex rules defining access controls and address space switching. As far as I

can tell, there is no compiler or linker support for these features, which are primarily used by high-performance database systems, so we won't address them further.

Instruction addressing on the 370 is also relatively straightforward. In the original 360, the jumps (always referred to as branch instructions) are all RR or RX format. In RR jumps, the second register operand contains the jump target, with register 0 meaning don't jump. In RX jumps, the memory operand is the jump target. The procedure call is branch and link (supplanted by the later branch and save for 31-bit addressing), which stores the return address in a specified register and then jumps to the address in the second register in the RR form or to the second operand address in the RX form.

For jumping around within a routine, the routine has to establish addressability, that is, a base register that points to (or at least close to) the beginning of the routine that RX instructions can use. By convention, register 15 contains the address of the entry point to a routine and can be used as a base register. Alternatively, an RR branch and link or branch and save with a second register of 0 stores the address of the subsequent instruction in the first operand register but doesn't jump; this can be used to set up a base register if the prior register contents are unknown. Since RX instructions have a 12-bit offset field, a single base register covers a 4KB piece of code. If a routine is bigger than that, it has to use multiple base registers to cover all of the routine's code.

The 390 added relative forms of most of the jumps. In these new forms, the instruction contains a signed 16-bit offset that is logically shifted left 1 bit (because instructions are aligned on even bytes) and that is added to the address of the instruction to get the address of the jump target. These new formats use no register to compute the address and permit jumps within $\pm 64\text{KB}$; this is enough for intraroutine jumps in all but the largest routines.

SPARC

SPARC has four major instruction formats and 31 minor instruction formats (Figure 2.5), four jump formats, and two data addressing modes.

SPARC comes close to living up to its billing as a reduced instruction set processor, although as the architecture has evolved through several versions, the original simple design has grown somewhat more complex. SPARC versions through V8 are 32-bit architectures. SPARC V9 expands the architecture to 64 bits.

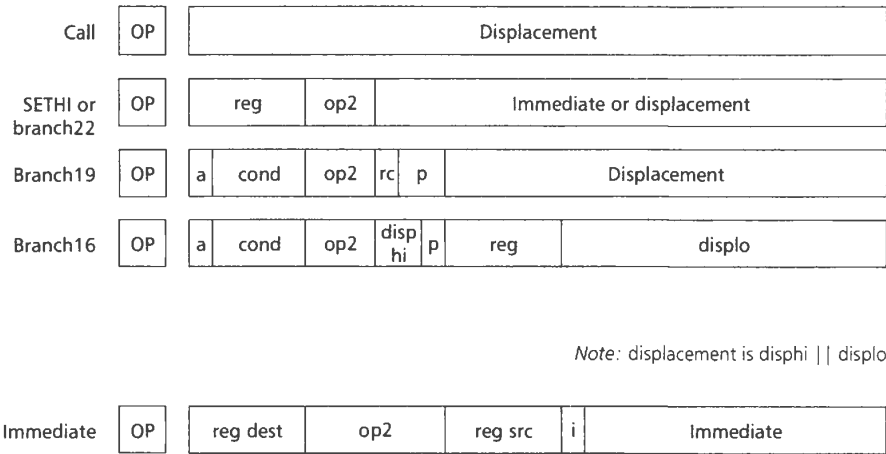


FIGURE 2.5 • SPARC.

SPARC V8

In SPARC V8, there are 31 general-purpose registers, each 32 bits, numbered from 1 to 31. Register 0 is a pseudo-register that always contains the value zero.

An unusual *register window* scheme attempts to minimize the amount of register saving and restoring at procedure calls and returns. The windows have little effect on linkers, so we won't discuss them further. (Register windows originated in the Berkeley RISC design from which SPARC is influenced.)

Data references use one of two addressing modes. One mode computes the address by adding the values in two registers together. (One of the registers can be register 0 if the other register already contains the desired address.) The other mode adds a 13-bit signed offset in the instruction to a base register.

SPARC assemblers and linkers support a pseudo-direct addressing scheme using a two-instruction sequence. The two instructions are SETHI, which loads its 22-bit immediate value into the high 22 bits of a register and zeros the lower 10 bits, followed by OR-immediate, which ORs its 13-bit immediate value into the low part of the register. The assembler and linker arrange to put the high and low parts of the desired 32-bit address into the two instructions.

The procedure call instruction and most conditional jump instructions (referred to as branches in SPARC literature) use relative addressing with various-size branch offsets ranging from 16 to 30 bits. Whatever the offset size,

the jump shifts the offset 2 bits left (because all instructions have to be at 4-byte word addresses), sign-extends the result to 32 or 64 bits, and adds that value to the address of the jump or call instruction to get the target address. The call instruction uses a 30-bit offset, which means it can reach any address in a 32-bit V8 address space. Calls store the return address in register 15. Various kinds of jumps use a 16-, 19-, or 22-bit offset, which is large enough to jump anywhere in any plausibly sized routine. The 16-bit format breaks the offset into a 2-bit high part and a 14-bit low part stored in different parts of the instruction word, but that doesn't cause any great trouble for the linker.

SPARC also has a jump and link instruction that computes the target address the same way that data reference instructions do, by adding together either two source registers or a source register and a constant offset. It also can store the return address in a target register.

Procedure calls use call or jump and link, which store the return address in register 15 and jump to the target address. Procedure return uses `JMP 8[r15]` to return two instructions after the call. (SPARC calls and jumps are delayed and optionally execute the instruction following the jump or call before jumping.)

SPARC V9

SPARC V9 expands all of the registers to 64 bits, using the low 32 bits of each register for old 32-bit programs. All existing instructions continue to work as before, except that register operands are now 64 rather than 32 bits. New load and store instructions handle 64-bit data, and new branch instructions can test either the 32- or 64-bit result of a previous instruction. SPARC V9 adds no new instructions for synthesizing full 64-bit addresses, nor is there a new call instruction. Full addresses can be synthesized by means of lengthy sequences that create the two 32-bit halves of the address in separate registers using `SETHI` and `OR`, shift the high-half 32 bits to the left, and `OR` the two parts together. In practice, 64-bit addresses are loaded from a pointer table, and intermodule calls load the address of the target routine from the table into a register and then use jump and link to make the call.

INTEL x86

The Intel x86 architecture is by far the most complex of the three that we discuss. It features an asymmetrical instruction set and segmented addresses.

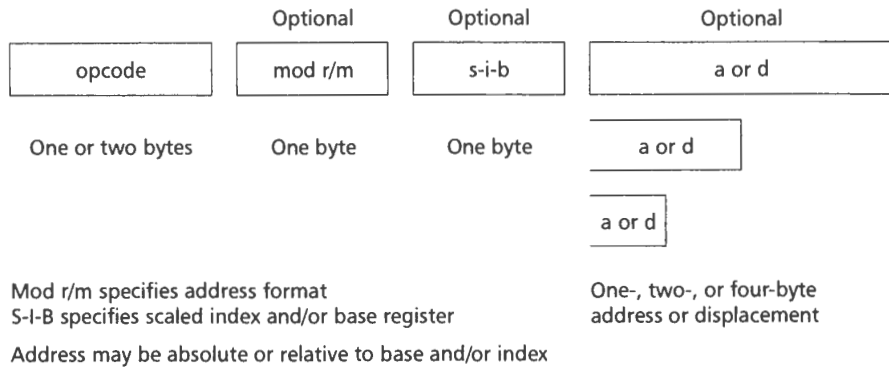


FIGURE 2.6 • Generalized x86 instruction format.

There are six 32-bit general-purpose registers named EAX, EBX, ECX, EDX, ESI, and EDI; two 32-bit registers used primarily for addressing, EBP and ESP; and six specialized 16-bit segment registers, CS, DS, ES, FS, GS, and SS. The low half of each of the 32-bit registers can be used as 16-bit registers, called AX, BX, CX, DX, SI, DI, BP, and SP, respectively. The low and high bytes of each of the AX through DX registers are 8-bit registers called AL, AH, BL, BH, CL, CH, DL, and DH. On the 8086, 186, and 286 chips, many instructions required their operands to be in specific registers, but on the 386 and later chips, most but not all of the functions that required specific registers have been generalized to use any register. The ESP is the hardware stack pointer and always contains the address of the current stack. The EBP pointer is usually used as a frame register that points to the base of the current stack frame. (The instruction set encourages but doesn't require this.)

At any moment an x86 is running in one of three modes: real mode, which emulates the original 16-bit 8086; 16-bit protected mode, which was added on the 286; or 32-bit protected mode, which was added on the 386. Here we primarily discuss 32-bit protected mode. Protected mode involves the x86's notorious segmentation, but we'll discuss that subject later (Section 2.8).

Most instructions that address data in memory use a common instruction format (Figure 2.6). (The ones that don't use specific architecture-defined registers; for example, the push and pop instructions always use ESP to address the stack.) Addresses are calculated by adding together any or all of a signed 1-, 2-, or 4-byte displacement value in the instruction; a base register, which can be any of the 32-bit registers; and an optional index register, which

can be any of the 32-bit registers except ESP. The index can be logically shifted left 0, 1, 2, or 3 bits to make it easier to index arrays of values.

Although it's possible for a single instruction to include all of the displacement, base, and index, most just use a 32-bit displacement, which provides direct addressing, or a base with a 1- or 2-byte displacement, which provides stack addressing and pointer dereferencing. From a linker's point of view, direct addressing permits an instruction or data address to be embedded anywhere in the program on any byte boundary.

Conditional and unconditional jumps and subroutine calls all use relative addressing. Any jump or call instruction can have a 1-, 2-, or 4-byte offset, which is added to the address of the instruction following the instruction to get the target address. This permits jumps and calls anywhere in the current 32-bit address space. Unconditional jumps and calls also can compute the target address using the full data address calculation described above; most often this is used to jump or call to an address stored in a register. Call instructions push the return address on the stack pointed to by ESP. Unconditional jumps and calls can also have a full 6-byte segment/offset address in the instruction or can calculate the address at which the segment/offset target address is stored. These call instructions push both the return address and the caller's segment number, to permit intersegment calls and returns.

2.7 PAGING AND VIRTUAL MEMORY

On most modern computers, each program can potentially address a vast amount of memory (4GB on a typical 32-bit machine). Few computers actually have that much memory, and even the ones that do need to share it among multiple programs. Paging hardware divides a program's address space into fixed-size *pages*, typically 2KB or 4KB in size, and divides the physical memory of the computer into *page frames* of the same size. The hardware contains *page tables* with an entry for each page in the address space, as shown in Figure 2.7.

A page table entry can contain the real memory page frame for the page, or it can contain flag bits to mark the page as not present. When an application program attempts to use a page that is not present, hardware generates a *page fault*, which is handled by the operating system. The operating system can load a copy of the contents page from disk into a free page frame, then let the

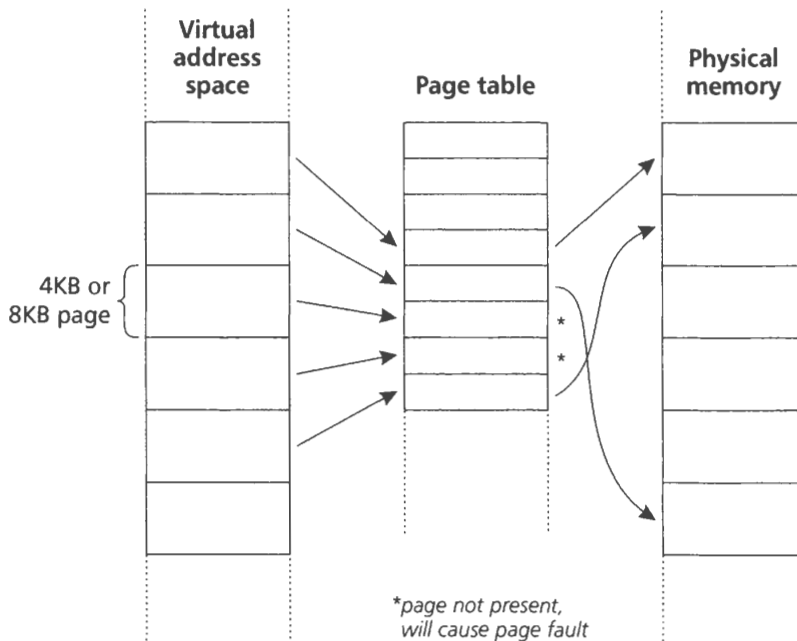


FIGURE 2.7 • Page mapping.

application continue. By moving pages back and forth between main memory and disk as needed, the operating system can provide *virtual memory*, which appears to the application to be far larger than the real memory in use.

Virtual memory comes at a cost, however. Individual instructions execute in a fraction of a microsecond, but a page fault and consequent page in or page out (transfer from disk to main memory or vice versa) takes several milliseconds because it requires a disk transfer.

The more page faults a program generates, the slower it runs, with the worst case being thrashing—all page faults with no useful work getting done. The fewer pages a program needs, the fewer page faults it will generate. If the linker can pack related routines into a single page or a small group of pages, paging performance improves.

If pages can be marked as read-only, performance also improves. Read-only pages don't need to be paged out because they can be reloaded from wherever they came from originally. If identical pages logically appear in multiple address spaces (which often happens when multiple copies of the same

program are running), a single physical page suffices for all of the address spaces.

An x86 with 32-bit addressing and 4KB pages would need a page table with 2^{20} entries to map an entire address space. Because each page table entry is usually 4 bytes, this would make the page tables an impractical 4MB long. As a result, paged architectures page the page tables, with upper-level page tables that point to lower-level page tables that point to the actual page frames corresponding to virtual addresses. On the 370, each entry in the upper-level page table (called the segment table) maps 1MB of address space, so the segment table in 31-bit address mode may contain up to 2048 entries. Each entry in the segment table may be empty (in which case the entire segment is not present) or may point to a lower-level page table that maps the pages in that segment. Each lower-level page table has up to 256 entries, one for each 4KB chunk of address space in the segment. The x86 divides up its page tables similarly, although the boundaries are different. Each upper-level page table (called a page directory) maps 4MB of address space, so the upper-level page table contains 1024 entries. Each lower-level page table also contains 1024 entries, to map the 1024 4KB pages in the 4MB of address space corresponding to that page table. The SPARC architecture defines the page size as 4KB and has three levels of page tables rather than two.

The two- or three-level nature of page tables is invisible to applications with one important exception: The operating system can change the mapping for a large chunk of the address space (1MB on the 370, 4MB on the x86, 256KB or 16MB on SPARC) by changing a single entry in an upper-level page table, so for efficiency reasons the address space is often managed in chunks of that size by replacing individual second-level page table entries rather than by reloading the whole page table on process switches.

THE PROGRAM ADDRESS SPACE

Every application program runs in an address space defined by a combination of the computer's hardware and operating system. The linker or loader needs to create a runnable program that matches that address space.

The simplest kind of address space was that provided by PDP-11 versions of UNIX. The address space was 64KB, starting at location zero. The read-only code of the program was loaded at location zero, with the read/write data following the code. The PDP-11 had 8KB pages, so the data started on the 8KB

boundary after the code. The stack grew downward, starting at 64KB – 1, and as the stack and data grew, the respective areas were enlarged; if they met, the program ran out of space. UNIX on the VAX—the follow-on to the PDP-11—used a similar scheme. The first 2 bytes of every VAX UNIX program were zero (a register save mask indicating not to save anything). As a result, a null all-zero pointer was always valid, and if a C program used a null value as a string pointer, the zero byte at location zero was treated as a null string. As a result, a generation of UNIX programs in the 1980s contained hard-to-find bugs involving null pointers, and for many years, UNIX ports to other architectures provided a zero byte at location zero because it was easier than finding and fixing all the null pointer bugs.

UNIX systems put each application program in a separate address space, and they put the operating system in an address space logically separate from the applications. Other systems put multiple programs in the same address space, making the linker's and particularly the loader's job more complex because a program's actual load address isn't known until the program is about to be run.

DOS on x86 systems uses no hardware protection, so the system and running applications share the same address space. When the system runs a program, it finds the largest piece of free memory, which may be anywhere in the address space, loads the program into it, and starts it. IBM mainframe operating systems do roughly the same thing, loading a program into an available chunk of address space. In both cases, either the program loader or in some cases the program itself has to adjust to the location where the program is loaded.

MS Windows has an unusual loading scheme. Each program is linked to load at a standard starting address, but the executable program file contains relocation information. When MS Windows loads the program, it places the program at that starting address if possible, but it may load it somewhere else if the preferred address isn't available.

MAPPED FILES

Virtual memory systems move data back and forth between real memory and disk, paging data to disk when it doesn't fit in real memory. Originally, paging all went to anonymous disk space separate from the named files in the file system. Soon after the invention of paging, however, designers noticed that it was

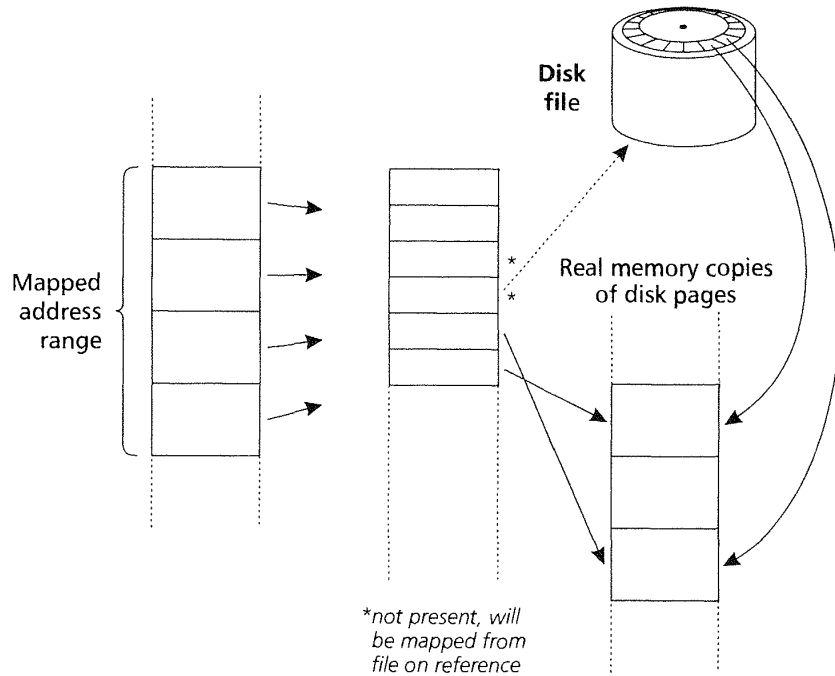


FIGURE 2.8 • Mapping a file.

possible to unify the paging system and the file system by using the paging system to read and write named disk files. When a program maps a file to a part of the program's address space, the operating system marks all of the pages in that part of the address space as not present and uses the file as the paging disk for that part of the address space (Figure 2.8). The program can read the file merely by referencing that part of the address space, at which point the paging system loads the necessary pages from disk.

There are three different approaches to handling writes to mapped files. The simplest is to map a file as read-only (RO), so that any attempts to store into the mapped region fail (usually causing the program to abort). The second is to map the file as read/write (RW), so that changes to the memory copy of the file are paged back to the disk by the time the file is unmapped. The third is to map the file as copy-on-write (COW, not the most felicitous acronym). This maps each page as read-only until the program attempts to store into the

page. At that time, the operating system makes a copy of the page, which is then treated as a private page that is not mapped from a file. From the program's point of view, mapping a file as COW is very similar to allocating a fresh area of anonymous memory and reading the file's contents into that area, because changes the program makes are visible to that program but not to any other program that might have mapped the same file.

SHARED LIBRARIES AND PROGRAMS

In nearly every system that handles multiple programs simultaneously, each program has a separate set of page tables, giving each program a logically separate address space. This makes a system considerably more robust, because buggy or malicious programs can't damage or spy on other programs, but it could potentially cause performance problems. If a single program or single program library is in use in more than one address space, the system can save a great deal of memory if all of the address spaces share a single physical copy of the program or library. This is relatively straightforward for the operating system to implement—just map the executable file into each program's address space. Unrelocated code and read-only data are mapped RO; writable data are mapped COW. The operating system can use the same physical page frames for RO and unwritten COW data in all the processes that map the file. (If the code has to be relocated at load time, the relocation process changes the code pages and they have to be treated as COW, not RO.)

Considerable linker support is needed to make this sharing work. In the executable program, the linker needs to group all of the executable code into one part of the file that can be mapped RO and the data into another part that can be mapped COW. Each section has to start on a page boundary, both logically in the address space and physically in the file. When several different programs use a shared library, the linker needs to mark each program so that when each starts, the library is mapped into the program's address space.

POSITION-INDEPENDENT CODE

When a program is in use in several different address spaces, the operating system can usually load the program at the same place in each of the address spaces in which it appears. This makes the linker's job much easier, because it

can bind all of the addresses in the program to fixed locations and no relocation needs to be done at the time the program is loaded.

Shared libraries complicate this situation considerably. In some simple shared-library designs, each library is assigned a globally unique memory address either at system boot time or at the time the libraries are created. This puts each library at a fixed address, but at the cost of creating a serious bottleneck to shared-library administration, because the global list of library memory addresses has to be maintained by the system manager. Furthermore, if a new version of a library appears that is larger than the previous version and doesn't fit into the address space assigned, then the entire set of shared libraries and, potentially, all of the programs that reference them may need to be relinked.

The alternative is to permit different programs to map a library to different places in the address space. This eases library administration, but the compiler, the linker, and the program loader need to cooperate so that the library will work regardless of where in the address space it appears.

One simple approach is to include standard relocation information with the library; when the library is mapped into each address space, the loader can fix up any relocatable addresses in the program to reflect the loaded addresses. Unfortunately, the process of fixing up the addresses involves writing into the library's code and data, which means that the pages will no longer be shared if they're mapped as COW or the program will crash if the pages are mapped as RO.

To avoid this problem, shared libraries use *position-independent code* (PIC), code that will work regardless of where in memory it is loaded. All the code in shared libraries is usually PIC, so the code can be mapped as read-only. Data pages still usually contain pointers that need relocation, but because data pages are mapped COW anyway, there's little sharing lost.

For the most part, PIC is pretty easy to create. All three of the architectures we have discussed in this chapter use relative jumps, so that jump instructions within the routines need no relocation. References to local data on the stack use based addressing relative to a base register, which doesn't need any relocation either. The only challenges are calls to routines not in the shared library and references to global data. Direct data addressing and the SPARC high/low register loading trick won't work, because they both require run-time relocation. Fortunately, there are a variety of tricks we can use to let PIC code handle interlibrary calls and global data. We discuss them when we cover shared libraries in detail in Chapters 9 and 10.

2.8 INTEL 386 SEGMENTATION

Next we turn to the notorious Intel architecture segmentation system. The x86 series is the only segmented architecture still in common use—other than some legacy ex-Burroughs Unisys mainframes—but because it's so popular, we have to deal with it. Although 32-bit operating systems don't make any significant use of segmentation, older systems and the very popular 16-bit embedded versions of the x86 series use it extensively.

The original 8086 was intended as a follow-on to Intel's quite popular 8-bit 8080 and 8085 microprocessors. The 8080 had a 16-bit address space, and the 8086 designers were torn between keeping the 16-bit address space—which made translation of 8085 programs easier and permitted more compact code—and providing a larger address space to give headroom for future applications in larger programs. They compromised by providing multiple 16-bit address spaces. Each 16-bit address space was known as a *segment*.

A running x86 program has four active segments defined by the four segment registers. The CS register defines the code segment, from which instructions are fetched. The DS register defines the data segment, from which most data are loaded and stored. The SS register defines the stack segment; it is used for the operands of push and pop instructions, the program address values pushed and popped by call and return instructions, and any data reference made using the EBP or ESP as a base register. The ES register defines the extra segment, used by a few string manipulation instructions. The 386 and later chips define two more segment registers, FS and GS. Any data reference can be directed into a specific segment by using a segment override. For example, the instruction `MOV EAX, CS:TEMP` fetches a data value from the location TEMP in the code segment rather than the data segment. The FS and GS segments are only used by means of segment overrides.

The segment values need not all be different. Most programs set the DS and SS values the same, so that pointers to stack variables and global variables can be used interchangeably. Some small programs set all four segment registers the same, providing a single address space known as the *tiny model*.

On the 8086 and 186, the architecture defined a fixed mapping from segment numbers to memory addresses by shifting the segment number 4 bits to the left. For example, segment number 0x123 would start at memory location 0x1230. This simple addressing is known as *real mode*. Programmers often

refer informally to *paragraphs*, 16-byte units of memory that a segment number can address.

The 286 added a protected mode, in which the operating system can map segments to arbitrary places in real memory and can mark segments as not present, providing segment-based virtual memory. Each segment can be marked executable, readable, or read/write, providing segment-level protection. The 386 extended protected mode to 32-bit addressing, so that each segment can be up to 4GB in size rather than only 64KB.

With 16-bit addressing, all but the smallest programs have to handle segmented addresses. Changing the contents of a segment register is quite slow (9 clock cycles on a 486 compared to 1 clock cycle to change the contents of a general-purpose register). As a result, programs and programmers go to great lengths to pack code and data into as few segments as possible to avoid having to change the contents of the segment registers. Linkers aid this process by providing “groups” that can collect related code or data into a single segment at link time. Code and data pointers can be either near (with an offset value but no segment number) or far (with both segment number and offset value).

Compilers can generate code for various memory models that determine whether code and data addresses are near or far by default. *Small model* code makes all pointers near and has one code and one data segment. *Medium model* code has multiple code segments (one per program source file) using far calls, but a single default data segment. *Large model* code has multiple code and data segments and all pointers are far by default. Writing efficient segmented code is very tricky and has been well documented elsewhere.

Segmented addressing places significant demands on the linker. Every address in a program has both a segment and an offset. Object files consist of multiple blocks of code that the linker packs into segments. Executable programs to be run in real mode have to mark all of the segment numbers that occur in the program so they can be relocated to the actual segments where the program is loaded. In addition, executable programs to be run in protected mode have to mark what data is to be loaded into what segment and the protection level (code, read-only data, read/write data) for each segment.

Although the 386 supports all of the 16-bit segmentation features of the 286, as well as 32-bit versions of all of the segmentation features, most 32-bit programs don't use segmentation at all. Paging, also added in the 386, provides most of the practical benefits of segmentation without the performance cost and the extra complications of writing segment manipulation code. Most

386 operating systems run applications in the tiny model, more often known as the *flat model* (a segment on a 386 is no longer tiny). They create a single code segment and a single data segment, each 4GB long, mapping them both to the full 32-bit paged address space. Even though the program is only using a single segment, that segment can be the full size of the address space.

The 386 makes it possible to use both 16-bit and 32-bit segments in the same program; a few operating systems, notably Windows 95 and 98, take advantage of that ability. Windows 95 and 98 run a lot of legacy Windows 3.1 code in 16-bit segments in a shared address space, while each new 32-bit program runs in its own flat model address space, with the 16-bit programs' address space mapped in to permit calls back and forth.

2.9 EMBEDDED ARCHITECTURES

Linking for embedded systems poses a variety of problems that rarely occur in other environments. Embedded chips have limited amounts of memory and limited performance, but because an embedded program may be built into chips in thousands or millions of devices, there are great incentives to make programs run as fast as possible using as little memory as possible. Some embedded systems use low-cost versions of general-purpose chips, such as the Intel 80186, while others use specialized processors such as the Motorola 56000 series of digital signal processors (DSPs).

ADDRESS SPACE QUIRKS

Embedded systems have small address spaces with quirky layouts. A 64KB address space can contain combinations of fast on-chip ROM and RAM, slow off-chip ROM and RAM, on-chip peripherals, and off-chip peripherals. There may be several noncontiguous areas of ROM or RAM. The 56000 has three address spaces of 64KB 24-bit words, each with combinations of RAM, ROM, and peripherals.

Embedded-chip development uses system boards that contain the processor chip along with supporting logic and chips. Frequently, different development boards for the same processor will have different memory layouts.

Different models of chips have differing amounts of RAM and ROM, so programmers have to trade off the effort to squeeze a program into a smaller memory versus the extra cost of using a more expensive version of the chip with more memory. A linker for an embedded system needs a way to specify the layout of the linked program in great detail, by assigning particular kinds of code or data or even individual routines and variables to specific addresses.

NON-UNIFORM MEMORY

References to on-chip memory are faster than those to off-chip memory, so in a system with both kinds, the most time-critical routines need to be placed in the fast memory. Sometimes it's possible to squeeze all of the program's time-critical code into the fast memory at link time. Other times it makes more sense to copy code or data from slow memory to fast memory as needed, so several routines can share the same fast memory at different times. To accomplish this trick, it's very useful to be able to tell a linker to "put this code at location XXXX but link it as though it's at location YYYY," so the code will be correct when it's copied from XXXX in slow memory to YYYY in fast memory at run time.

MEMORY ALIGNMENT

DSPs frequently have stringent memory alignment requirements for certain kinds of data structures. The Motorola 56000 series, for example, has an addressing mode to handle circular buffers very efficiently, so long as the base address of the buffer is aligned on a power-of-two boundary at least as large as the buffer size (for example, a 50-word buffer would need to be aligned on a 64-word boundary). The Fast Fourier Transform (FFT), an extremely important calculation for signal processing, depends on address bit manipulations that also require that the data on which an FFT operates be power-of-two aligned. Unlike on conventional architectures, the alignment requirements depend on the sizes of the data arrays, so that packing them efficiently into available memory can be tricky and tedious.

**EXERCISE • 2.1**

Use these SPARC program instructions to answer the following questions. (These instructions aren't intended as a useful program, just as some instruction format examples.)

Loc	Hex	Symbolic
1000	40 00 03 00	CALL X
1004	01 00 00 00	NOP ; no operation, for delay
1008	7F FF FE ED	CALL Y
100C	01 00 00 00	NOP
1010	40 00 00 02	CALL Z
1014	01 00 00 00	NOP
1018	03 37 AB 6F	SETHI r1,3648367 ; set high 22 bits of r1
101C	82 10 62 EF	ORI r1,r1,751 ; OR in low 10 bits of r1

- In a CALL instruction, the high 2 bits are the instruction code and the low 30 bits are a signed word (not byte) offset. What are the hex addresses for X, Y, and Z?
- What does the call to Z at location 1010 accomplish?
- The two instructions at 1018 and 101C load a 32-bit address into register 1. The SETHI loads the low 22 bits of the instruction into the high 22 bits of the register, and the ORI logically ORs the low 13 bits of the instruction into the register. What address will register 1 contain?
- If the linker moves X to be at location hex 2504 but doesn't change the location of the code in the example, to what will it change the instruction at location 1000 so that it still refers to X?

EXERCISE • 2.2

Use these Pentium program instructions to answer the following questions. Don't forget that the x86 is little-endian.

Loc	Hex	Symbolic
1000	E8 12 34 00 00	CALL A
1005	E8 ?? ?? ?? ??	CALL B
100A	A1 12 34 00 00	MOV %EAX,P
100F	03 05 ?? ?? ?? ??	ADD %EAX,Q

- At what location are routine A and data word P located? (*Tip: On the x86, relative addresses are computed relative to the byte address after the instruction.*)
- If routine B is located at address 0F00 and data word Q is located at address 3456, what are the byte values of the ?? bytes in the example?

EXERCISE • 2.3

Does a linker or loader need to understand every instruction in the target architecture's instruction set? If a new model of the target adds new instructions, will the linker need to be changed to support them? What if the model adds new addressing modes to existing instructions, as the 386 did relative to the 286?

EXERCISE • 2.4

Back in the Golden Age of computing, when programmers worked in the middle of the night because that was the only time they could get computer time, rather than because that's when they were awake, many computers used word rather than byte addresses. The PDP-6 and -10, for example, had 36-bit words and 18-bit addressing, in which each instruction was a word with the operand address in the low half of the word. (Programs could also store addresses in the high half of a data word, although there was no direct instruction set support for that.) How different is linking for a word-addressed architecture compared to linking for a byte-addressed architecture?

EXERCISE • 2.5

How hard would it be to build a retargetable linker, that is, one that could be built to handle different target architectures by changing a few specific parts of the source code for the linker? How about a multitarget linker, to handle code for a variety of different architectures (although not in the same linker job)?

8

LOADING AND OVERLAYS

Loading is the process of bringing a program into main memory so it can run. In this chapter we look at the loading process, concentrating on loading programs that have already been linked. Many systems used to have linking loaders that combined the linking and loading process, but those have now practically disappeared; the only ones I know of on current hardware are on MVS and the dynamic linkers we'll cover in Chapter 10. Linking loaders weren't all that different from plain linkers, with the primary and obvious difference being that the output was left in memory rather than placed in a file.

8.1 BASIC LOADING

We touched on most of the basics of loading in Chapter 3 in the context of object file design. Loading is a little different depending on whether a program is loaded by mapping into a process address space by means of the virtual memory system or just read in using normal I/O calls.

On most modern systems, each program is loaded into a fresh address space, which means that all programs are loaded at a known fixed address and

can be linked for that address. In that case, loading is pretty simple and requires the following steps:

1. Read enough header information from the object file to find out how much address space is needed.
2. Allocate that address space, in separate segments if the object format has separate segments.
3. Read the program into the segments in the address space.
4. Zero out any bss space at the end of the program if the virtual memory system doesn't do so automatically.
5. Create a stack segment if the architecture needs one.
6. Set up any run-time information such as program arguments or environment variables.
7. Start the program.

If the program isn't mapped through the virtual memory system, reading in the object file just means reading in the file with normal read system calls. On systems that support shared read-only code segments, the system needs to check whether there's already a copy of the code segment loaded in and uses that rather than making another copy.

On systems that do memory mapping, the process is slightly more complicated. The system loader has to create the segments and then arrange to map the file pages into the segments with appropriate permissions, either read-only (RO) or copy-on-write (COW). In some cases, the same page is double mapped at the end of one segment and the beginning of the next as RO in one and COW in the other (in formats like compact UNIX a.out). The data segment is generally contiguous with the bss segment, so the loader has to zero out the part of the last page after the end of the data (because the disk version usually has symbols or something else there) and allocate enough zero pages following the data to cover the bss segment.

8.2 BASIC LOADING, WITH RELOCATION

A few systems still do load-time relocation for executables, and many do load-time relocation of shared libraries. Some, like DOS, lack usable hardware

relocation. Others, like MVS, have hardware relocation but are descended from systems that didn't have it. Some systems have hardware relocation but can load multiple executable programs and shared libraries into the same address space, so linkers can't count on having specific addresses available.

As discussed in Chapter 7, load-time relocation is far simpler than link-time relocation, because the entire program is relocated as a unit. If, for example, the program is linked as though it would be loaded at location zero but is in fact loaded at location 15000, all of the places in the program that require fixups will get 15000 added. After reading the program into memory, the loader consults the relocation items in the object file and fixes up the memory locations to which the items point.

Load-time relocation can present a performance problem, because code loaded at different virtual addresses can't usually be shared between address spaces (the fixups for each address space are different). One approach, used by MVS and to some extent by MS Windows and AIX, is to create a shared-memory area that is present in multiple address spaces and to load often-used programs into that. (MVS calls this the link pack area.) This has the problem that different processes don't get separate copies of writable data, so the application has to be written to allocate all of its writable storage explicitly.

8.3 POSITION-INDEPENDENT CODE

One popular solution to the dilemma of loading the same program at different addresses is *position-independent code* (PIC). The idea is simple: separate the code from the data and generate code that won't change regardless of the address at which it's loaded. That way the code can be shared among all processes, with only data pages being private to each process.

This is a surprisingly old idea. TSS/360 used it in 1966, and I don't believe it was original there. (TSS was notoriously buggy, but I can report from personal experience that the PIC features really worked.)

On modern architectures, it's not difficult to generate PIC-executable code. Jumps and branches are generally either pc-relative or relative to a base register set at run time, so no load-time relocation is required for them. The problem is with data addressing. The code can't contain any direct data addresses, because those would be relocatable and wouldn't be PIC. The usual solution is to create a table of data addresses in a data page and to keep a pointer to that table in a register, so the code can use indexed addressing

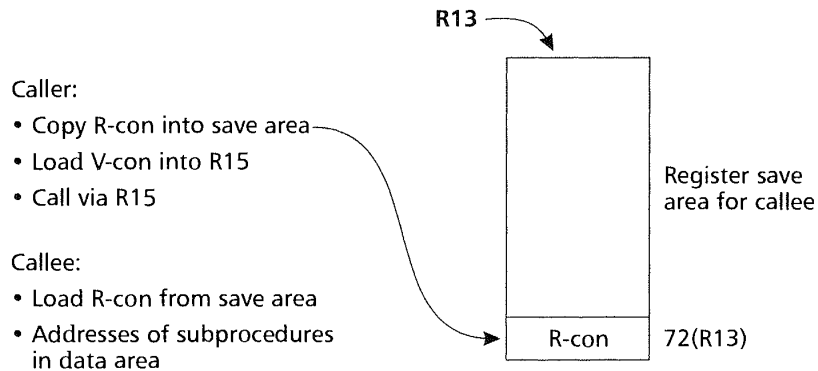


FIGURE 8.1 • TSS-style two-address procedure call.

relative to that register to pick up the data addresses. This works at the cost of an extra indirection for each data reference, but there's still the question of how to get the initial data address into the register.

TSS/360 POSITION-INDEPENDENT CODE

TSS took a brute-force approach. Every routine had two addresses: the address of the code, known as the V-con (short for V-style address constant, which even non-PIC code needed), and the address of the data, known as the R-con. The standard OS/360 calling sequence requires that the caller provide an 18-word register save area pointed to by register 13. TSS extended the save area to 19 words and required that the caller place the callee's R-con into that 19th word at 72(R13) before making the call (Figure 8.1). Each routine had in its data segment the V-cons and R-cons for all of the routines that it called and stored the appropriate R-con into the outgoing save area before each call. The main routine in a program received a save area from the operating system that provided the initial R-con.

This scheme worked, but it is poorly suited for modern systems. For one thing, copying the R-cons made the calling sequence bulky. For another, it made procedure pointers two words, which didn't matter in the 1960s but is an issue now because in programs written in C, all pointers have to be the same size. (The C standard doesn't mandate it, but far too much existing C code assumes it to do anything else.)

Caller:

- Load pointer table address into RP
- Load code address from 0(RP) into RC
- Call via RC

Callee:

- RP points to pointer table
- Table has addresses of pointer tables for subprocedures

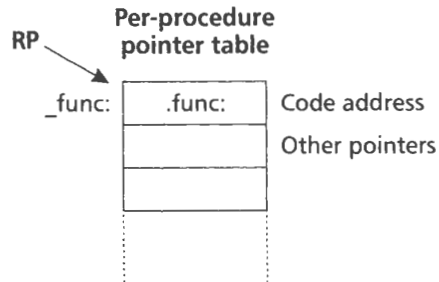


FIGURE 8.2 • Accessing code by means of data pointers.

PER-ROUTINE POINTER TABLES

A simple modification used in some UNIX systems is to treat the address of a procedure's data as the address of the procedure and to place a pointer to the procedure's code at that address (Figure 8.2). To call a procedure, the caller loads the data address into an agreed data pointer register, then it loads the code address from the location pointed to by the data pointer into a scratch register and calls the routine. This is easy to implement and has adequate if not fabulous performance.

TABLE OF CONTENTS

IBM's AIX uses a more sophisticated version of this scheme. AIX programs group routines into *modules*, with a module typically being the object code generated from a single C or C++ source file or a group of related source files. The data segment of each module contains a table of contents (TOC), which contains the combined pointer tables for all of the routines in the module as well as some of the small static data for the routines. Register 2 always contains the address of the TOC for the current module, permitting direct access to the static data in the TOC and indirect addressing of code and data to which the TOC contains pointers. Calls within a single module are a single call instruction, since the caller and callee share the same TOC. Intermodule calls have to switch TOCs before the call and switch back afterward.

Compilers generate each call as a call instruction, followed by a placeholder no-op instruction, which is correct for intramodule calls. When the

linker encounters an intermodule call, it generates a routine called a global linkage or glink at the end of the module's text segment. The glink saves the caller's TOC on the stack, loads the callee's TOC and address from pointers in the caller's TOC, then jumps to the routine. The linker redirects each intermodule call to the glink for the called routine and patches the following no-op to a load instruction that restores the TOC from the stack. Procedure pointers are pointers to a TOC/code pair, and calls through a pointer use a generic glink routine that uses the TOC and the code address that the pointer points to.

This scheme makes intramodule calls as fast as possible. Intermodule calls are slowed down somewhat by the detour through the glink routine, but the slowdown is small compared to some of the alternatives we'll see in the next section.

ELF POSITION-INDEPENDENT CODE

UNIX System V Release 4 introduced a PIC scheme similar to the TOC scheme for its ELF shared libraries. This scheme is now universally used by systems that use ELF executables. It has the advantage of returning to the normal convention that the address of a procedure is the address of the code for the procedure, regardless of whether you are calling PIC code (found in shared ELF libraries) or non-PIC code (found in regular ELF executables); it works at the cost of somewhat more per-routine overhead than in the TOC scheme.

Its designers noticed that an ELF executable consists of a group of code pages followed by a group of data pages, and regardless of where in the address space the program is loaded, the offset from the code to the data doesn't change. So if the code can load its own address into a register, the data will be at a known distance from that address, and references to data in the program's own data segment can use efficient based addressing with fixed offsets (Figure 8.3).

The linker creates a global offset table (GOT) that contains pointers to all of the global data that the executable file addresses. (Each shared library has its own GOT, and if the main program is compiled with PIC—which it normally isn't—it will have a GOT as well.) Because the linker creates the GOT, there is only one pointer per ELF executable for each datum, regardless of how many routines in the executable refer to it.

If a procedure needs to refer to global or static data, it's up to the procedure itself to load up the address of the GOT. The details vary by architecture, but the following 386 code is typical:

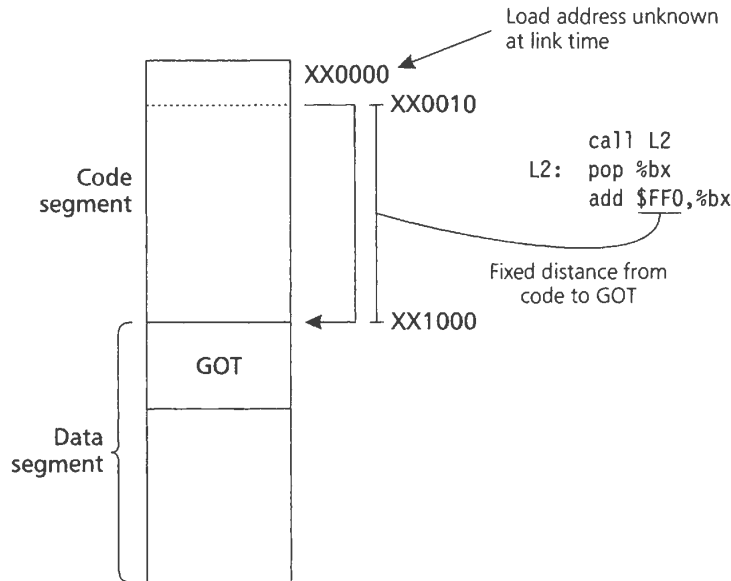


FIGURE 8.3 • PIC code and data with fixed offsets.

```
call .L2    ;; push PC in on the stack
.L2:
popl %ebx   ;; PC into register EBX
addl $_GLOBAL_OFFSET_TABLE_[.-.L2],%ebx ;; adjust ebx to GOT address
```

This code consists of a call instruction to the immediately following location—which has the effect of pushing the PC on the stack but not jumping—then a pop to get the saved PC in a register and an add-immediate of the difference between the address of the GOT and the address of the target of the call. In an object file generated by a compiler, there's a special `R_386_GOTPC` relocation item for the operand of the `addl` instruction. This item tells the linker to substitute in the offset from the current instruction to the base address of the GOT, and it also serves as a flag to the linker to build a GOT in the output file. In the output file, there's no relocation needed for the instruction because the distance from the `addl` to the GOT is fixed.

Once the GOT register is loaded, code can reference local static data using the GOT register as a base register, because the distance from a static datum in the program's data segment to the GOT is fixed at link time. Addresses of

global data aren't bound until the program is loaded (see Chapter 10), so to reference global data, code has to load a pointer to the data from the GOT and then dereference the pointer. This extra memory reference makes programs somewhat slower, although it's a cost that most programmers are willing to pay for the convenience of dynamically linked libraries. Speed-critical code can use static shared libraries (see Chapter 9) or no shared libraries at all.

To support PIC, ELF defines a handful of special relocation types for code that uses the GOT in addition to `R_386_GOTPC` or its equivalent. The exact types are architecture specific, but the following x86 types are typical:

- `R_386_GOT32`. This is the relative location of the slot in the GOT where the linker has placed a pointer to the given symbol. It is used for indirectly referenced global data.
- `R_386_GOTOFF`. This is the distance from the base of the GOT to the given symbol or address. It is used to address static data relative to the GOT.
- `R_386_RELATIVE`. This is used to mark data addresses in a PIC shared library that need to be relocated at load time.

For example, consider this piece of C code:

```
static int a; /* static variable */
extern int b; /* global variable */
...
a = 1; b = 2;
```

Variable `a` is allocated in the bss segment of the object file, which means that it is at a known fixed distance from the GOT. Object code can reference this variable directly, using the `ebx` as a base register and a GOT-relative offset:

```
movl $1,a@GOTOFF(%ebx) ;; R_386_GOTOFF reference to variable "a"
```

Variable `b` is global, and its location may not be known until run time if it turns out to be in a different ELF library or executable. In this case, the object code references a pointer to `b`, which the linker creates in the GOT:

```
movl b@GOT(%ebx),%eax ;; R_386_GOT32 ref to address of "b"
movl $2, (%eax)
```

Note that the compiler only creates the `R_386_GOT32` reference, and it's up to the linker to collect all such references and make slots for them in the GOT.

Finally, ELF shared libraries contain `R_386_RELATIVE` relocation entries that the run-time loader, part of the dynamic linker we examine in Chapter 10, uses to do load-time relocation. Because the text in shared libraries is invariably PIC, there are no relocation entries for the code, but data can't be PIC, so there is a relocation entry for every pointer in the data segment. (Actually, you can build a shared library with non-PIC code, in which case there will be relocation entries for the text as well, although almost nobody does that because it makes the text nonsharable.)

PIC COSTS AND BENEFITS

The advantages of PIC are straightforward; PIC makes it possible to load code without having to do load-time relocation and to share memory pages of code among processes even though they don't all have the same address space allocated. The possible disadvantages are slowdowns at load time, in procedure calls, and in function prologue and epilogue, as well as overall slower code.

At load time, although the code segment of a PIC file need not be relocated, the data segment does need to be relocated. In large libraries, the TOC or GOT can be very large and it can take a long time to resolve all the entries. This is as much a problem with dynamic linking, which we'll address in Chapter 10, as with PIC. Handling `R_386_RELATIVE` items or the equivalent to relocate GOT pointers to data in the same executable is fairly fast, but the problem is that many GOT entries point to data in other executables and require a symbol table lookup to resolve.

Calls in ELF executables are usually dynamically linked, even calls within the same library, which adds significant overhead. We examine this in Chapter 10.

Function prologues and epilogues in ELF files are quite slow. They have to save and restore the GOT register—`ebx` in the x86—and the dummy call and pop to get the program counter into a register are quite slow. From a performance viewpoint, the TOC approach used in AIX wins here, because each procedure can assume that its TOC register is already set at procedure entry.

Finally, PIC code is bigger and slower than non-PIC code. The slowdown varies greatly by architecture. On RISC systems with plenty of registers and no direct addressing, the loss of one register as the TOC or GOT pointer isn't significant, and lacking direct addressing they need a constant pool of some sort

anyway. The worst case is on the x86: It only has six registers, so losing one of them to be the GOT pointer can make code significantly worse. Since the x86 does have direct addressing, a reference to external data that would be a simple MOV or ADD instruction in non-PIC code turns into a load of the address followed by the MOV or ADD, which both adds an extra memory reference and uses yet another precious register for the temporary pointer.

Particularly on x86 systems, the performance loss in PIC code is significant in speed-critical tasks, enough so that some systems retreat to a sort-of-PIC approach for shared libraries. We'll revisit this issue in Chapters 9 and 10.

8.4 BOOTSTRAP LOADING

Up to this point, the discussions of loading have all presumed that there's already an operating system or at least a program loader resident in the computer to load the program of interest. The chain of programs being loaded by other programs has to start somewhere, so the obvious question is, how is the first program loaded into the computer?

In modern computers, the first program the computer runs after a hardware reset invariably is stored in a ROM area known as the *bootstrap*, or *boot*, *ROM*. When the CPU is powered on or reset, it sets its registers to a known state. On x86 systems, for example, the reset sequence jumps to the address 16 bytes below the top of the system's address space. The bootstrap ROM occupies the top 64KB of the address space; ROM code then starts up the computer. On IBM-compatible x86 systems, the boot ROM code reads the first block of the floppy disk—or if that fails, the first block of the first hard disk—into memory location zero and then jumps to location zero. The program in block zero in turn loads a slightly larger operating system boot program from a known place on the disk into memory, and it jumps to that program, which in turn loads in the operating system and starts it. (There can be even more steps such as a boot manager that decides from which disk partition to read the operating system boot program, but the sequence of increasingly capable loaders remains.)

Why not just load the operating system directly? The answer here is that you can't fit an operating system loader into 512 bytes. The first-level loader typically is only able to load a single-segment program from a file with a fixed name in the top-level directory of the boot disk. The operating system loader

contains more sophisticated code that can read and interpret a configuration file, uncompress a compressed operating system executable, and address large amounts of memory. (On an x86 the loader usually runs in real mode, which means that it's tricky to address more than 1MB of memory.) The full operating system can turn on the virtual memory system, load the drivers it needs, and then proceed to run user-level programs.

Many UNIX systems use a similar bootstrap process to get user-mode programs running. The kernel creates a process and then inserts a tiny program, only a few dozen bytes long, into that process. The tiny program executes a system call that runs `/etc/init`, the user-mode initialization program that in turn runs configuration files and starts the daemons and login programs that a running system needs.

None of this matters much to the application-level programmer, but it becomes more interesting if you want to write programs that run on the bare hardware of the machine, because then you need to arrange to intercept the bootstrap sequence somewhere in order to run your program rather than the usual operating system. Some systems make this quite easy (just stick the name of your program in `AUTOEXEC.BAT` and reboot Windows 95, for example); others make it nearly impossible. It also presents opportunities for customized systems. For example, a single-application system could be built over a UNIX kernel by naming the application `/etc/init`.

8.5 TREE-STRUCTURED OVERLAYS

The final topic of this chapter is tree-structured overlays, an approach that was widely used in the days before there was virtual memory to fit a program into a memory area smaller than the program. Overlays are another technique that dates back to before 1960 and are still in use in some memory-constrained environments. Several DOS linkers in the 1980s supported them in a form nearly identical to that used 25 years earlier on mainframe computers. Although overlays are now little used on conventional architectures, the techniques that linkers use to create and manage overlays remain interesting. Also, the intersegment call tricks developed for overlays point the way to dynamic linking. In environments like DSPs that have constrained program address spaces, overlay techniques can be a good way to squeeze programs in, especially because overlay managers tend to be small. The OS/360 overlay

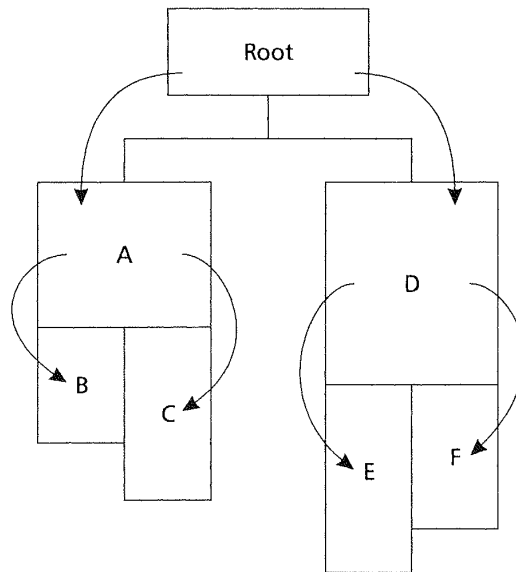


FIGURE 8.4 • Typical overlay tree format.

manager is only about 500 bytes, and I once wrote one for a graphics processor with a 512-word address space that used only a dozen words or so.

Overlay programs divide the code into a tree of segments, such as the one in Figure 8.4. The programmer manually assigns object files or individual object code segments to overlay segments. Sibling segments in the overlay tree share the same memory. In the example in Figure 8.4, segments A and D share the same memory, B and C share the same memory, and E and F share the same memory. The sequence of segments that lead to a specific segment is called a *path*, so the path for E includes the root, D, and E.

When the program starts, the system loads the root segment, which contains the entry point of the program. Each time a routine makes a downward intersegment call, the overlay manager ensures that the path to the call target is loaded. For example, if the root calls a routine in segment A, the overlay manager loads segment A if it's not already loaded. If a routine in A calls a routine in B, the manager has to ensure that B is loaded; if a routine in the root calls a routine in B, the manager has to ensure that both A and B are loaded. Upward calls don't require any linker help, because the entire path from the root is already loaded.

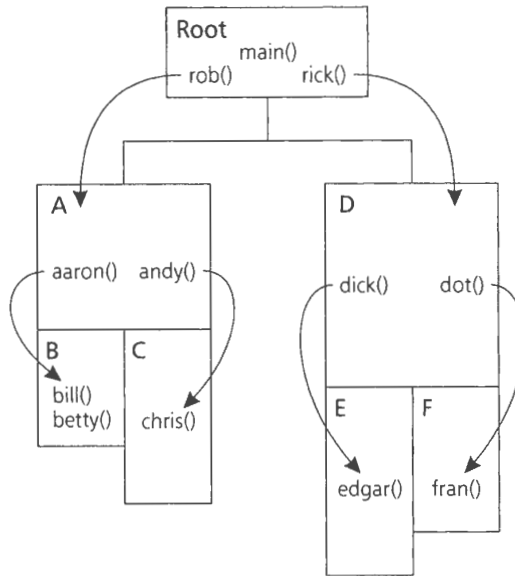


FIGURE 8.5 • Overlay tree example.

Calls across the tree are known as *exclusive calls* and are usually considered to be errors because it's not possible to return. Overlay linkers let the programmer force exclusive calls for situations where the called routine is known not to return.

DEFINING OVERLAYS

Overlay linkers create overlay executables from ordinary input object files. The objects don't contain any overlay instructions; instead, the programmer specifies the overlay structure with a command language that the linker reads and interprets. Figure 8.5 shows the same overlay structure as before with the names of the routines loaded into each segment.

Figure 8.6 shows the linker commands that you might give to the IBM 360 linker to create this structure. Spacing doesn't matter, so we've indented the commands to show the tree structure. OVERLAY commands define the beginning of each segment; commands with the same overlay name define segments that overlay each other. Hence the first OVERLAY AD defines segment A,

```

INCLUDE ROB
INCLUDE RICK
OVERLAY AD
    INCLUDE AARON, ANDY
    OVERLAY BC
        INCLUDE BILL, BETTY
    OVERLAY BC
        INCLUDE CHRIS
OVERLAY AD
    INCLUDE DICK, DOT
    OVERLAY EF
        INCLUDE EDGAR
    OVERLAY EF
        INCLUDE FRAN

```

FIGURE 8.6 • Linker commands corresponding to overlay tree example in Figure 8.5.

and the second defines segment D. Overlay segments are defined in a depth-first left-to-right tree walk. INCLUDE commands name logical files for the linker to read.

It's up to the programmer to lay out overlays to be space efficient. The storage allocated for each segment is the maximum length of any of the segments that occupy the same space. For example, assume that the file lengths in decimal are as follows:

<i>Name</i>	<i>Size</i>	<i>Name</i>	<i>Size</i>
rob	500	chris	3000
rick	1500	dick	3000
aaron	3000	dot	4000
andy	1000	edgar	2000
bill	1000	fran	3000
betty	1000		

The corresponding storage allocation looks like Figure 8.7. Each segment starts immediately after the preceding segment in the path, and the total program size is the length of the longest path. This program is fairly well

	0 rob		
	500 rick		
2000 aaron		2000 dick	
5000 andy		5000 dot	
6000 bill	6000 chris		
7000 betty	9000 —	9000 edgar	9000 fran
8000 —		11000 —	12000 —

FIGURE 8.7 • Overlay storage allocation example.

balanced, with the longest path being 12000 and the shortest being 8000. Juggling the overlay structure to find one that is as compact as possible while still being valid (no exclusive calls) and reasonably efficient is a black art requiring considerable trial and error. Since the overlays are defined entirely in the linker, each trial requires a relink but no recompilation.

IMPLEMENTATION OF OVERLAYS

The implementation of overlays is surprisingly simple. Once the linker determines the layout of the segments, it relocates the code in each segment appropriately based on the memory location of the segment. The linker needs to create a segment table that goes in the root segment and, in each segment, glue code (linker-generated code that hooks into the overlay manager) for each routine that is the target of a downward call from that segment. The segment table lists each segment, a flag to note if the segment is loaded, the segment's path, and information needed to load the segment from disk (Figure 8.8).

The linker interposes the glue code in front of each downward call so that the overlay manager can ensure that the required segment(s) are loaded. Segments can use glue code in higher-level but not lower-level routines. For example, if routines in the root call aaron, dick, and betty, the root needs glue code for each of those three symbols. If segment A contains calls to bill, betty, and chris, segment A needs glue code for bill and chris but can use the glue for betty already present in the root. All downward calls (which are to global symbols) are resolved to glue code (Figure 8.9) rather than to the actual routine.

```

struct segtab {
    struct segtab *path; // preceding segment in path
    boolean ispresent;   // true if this segment is loaded
    int memoffset;       // relative load address
    int diskoffset;      // location in executable
    int size;            // segment size
} segtab[];

```

FIGURE 8.8 • Idealized segment table.

The glue code has to save any registers it changes, because it has to be transparent to the calling and called routine, then it has to jump into the overlay manager, providing the address of the real routine and an indication of which segment that address is in. Here we use a pointer, but an index into the segtab array would work as well.

At run time, the system loads in the root segment and starts it. At each downward call, the glue code calls the overlay manager. The manager checks the target segment's status. If the segment is present, the manager just jumps to the real routine. If the segment is not present, the manager loads the target segment and any unloaded preceding segments in the path, marks any conflicting segments as not present, marks the newly loaded segments as present, and jumps.

OVERLAY FINE POINTS

As always, details make elegant tree-structured overlays messier than they might be.

Data

We've been talking about structuring code overlays, without any consideration of where the data go. Individual routines may have private data loaded into the segments with the routines, but any data that have to be remembered from one call to the next need to be promoted high enough in the tree that they won't get unloaded and reloaded, which would lose any changes made. In practice, this means that most global data usually end up in the root. When

```

glue'betty: call load_overlay
      .long betty      // address of real routine
      .long segtab+N  // address of segment B's segtab

```

FIGURE 8.9 • Idealized glue code for the x86.

Fortran programs are overlaid, overlay linkers can position common blocks appropriately to be used as communication areas. For example, if dick calls edgar and fran, and the latter two both refer to a common block, that block has to reside in segment D to be a communication area.

Duplicated Code

The overall structure of an overlay program frequently can be improved by duplicating code. In our example, imagine that chris and edgar both call a routine called greg that is 500 bytes long. A single copy of greg would have to go in the root, increasing the total loaded size of the program, because placing it anywhere else in the tree would require a forbidden exclusive call from either chris or edgar. On the other hand, if both segments C and E include copies of greg, the overall loaded size of the program doesn't increase, because the end of segment C would grow from 9000 to 9500 and of segment E from 11000 to 11500, both of which are still smaller than the 12000 bytes that F requires.

Multiple Regions

Frequently, a program's calling structure doesn't map very well to a single tree. Overlay systems handle multiple code regions, with a separate overlay tree in each region. Calls between regions, other than to root segments, always go through glue code. The IBM linker supports up to four regions, although in my experience I have never found a use for more than two.

OVERLAY SUMMARY

Even though overlays have been rendered largely obsolete by virtual memory, they remain of historical interest because they were the first significant use of link-time code generation and modification. They required a great deal of

manual programmer work to design and specify the overlay structure, generally with a lot of trial-and-error “digital origami,” but they were a very effective way to squeeze a large program into limited memory.

Overlays originated the important technique of wrapping call instructions in the linker to turn a simple procedure call into one that does more work (in this case, loading the required overlay). Linkers have used wrapping in a variety of ways. The most important is dynamic linking, which we cover in Chapter 10, where it is used to link to a called routine in a library that may not have been loaded yet. Wrapping is also useful for testing and debugging, in order to insert checking or validation code in front of a suspect routine without changing or recompiling the source file.

EXERCISES

EXERCISE • 8.1

Compile some small C routines with PIC and non-PIC code. How much slower is the PIC code than the non-PIC code? Is it slower enough to be worth having non-PIC versions of libraries for programmers who are in a hurry?

EXERCISE • 8.2

In the overlay example, assume that dick and dot each call both edgar and fran, but dick and dot don’t call each other. Restructure the overlay so that dick and dot share the same space, and adjust the structure so that the call tree still works. How much space does the overlay program take now?

EXERCISE • 8.3

In the overlay segment table, there’s no explicit marking of conflicting segments. When the overlay manager loads a segment and the segment’s path, how does the manager determine what segments to mark as not present?

EXERCISE • 8.4

In an overlay program with no exclusive calls, is it possible that a series of calls could end up jumping to unloaded code anyway? In the example in this chapter, what happens if rob calls bill, which calls aaron, which calls chris, and then the routines all return? How hard would it be for the linker or overlay manager to detect or prevent that problem?

**PROJECT • 8.1**

Add a feature to the linker to wrap routines. Create a linker switch

`-w name`

that wraps the given routine. Change all references to the named routine in the program to be references to `wrap_name`. (Be sure not to miss internal references within the segment in which the name is defined.) Change the name of the routine to `real_name`. (This lets the programmer write a wrapper routine called `wrap_name` that can call the original routine as `real_name`.)

PROJECT • 8.2

Starting with the linker skeleton from Chapter 3, write a tool that modifies an object file to wrap a name, that is, that causes references to `name` to turn into external references to `wrap_name` and the existing routine to be renamed `real_name`. Why would we want to use such a program rather than building the feature into the linker? (*Hint:* Consider the case where you're not the author or maintainer of the linker.)

PROJECT • 8.3

Add support to the linker to produce executables with position-independent code. Add the following new 4-byte relocation types:

```
loc seg ref GA4
loc seg ref GP4
loc seg ref GR4
loc seg ref ER4
```

The types have the following roles:

- GA4 (GOT address). At location `loc`, store the distance to the GOT.
- GP4 (GOT pointer). Put a pointer to symbol `ref` in the GOT, and at location `loc`, store the GOT-relative offset of that pointer.
- GR4 (GOT relative). Location `loc` contains an address in segment `ref`; replace that with the offset from the beginning of the GOT to that address.
- ER4 (Executable relative). Location `loc` contains an address relative to the beginning of the executable. The `ref` field is ignored.

In your linker's first pass, look for GP4 relocation entries, build a GOT segment with all the required pointers, and allocate the GOT segment just before the data and bss segments. In the second pass, handle the GA4, GP4, and GR4 entries. In the output file, create ER4 relocation entries for any data that would have to be relocated if the output file were loaded at other than its nominal address. This would include anything marked by an A4 or AS4 relocation entry in the input. (*Hint: Don't forget the GOT.*)

SHARED LIBRARIES

Program libraries date back to the earliest days of computing, when programmers quickly realized that they could save a lot of time and effort by reusing blocks of program code. With the advent of compilers for languages like Fortran and COBOL, libraries became an integral part of programming. Compiled languages use libraries explicitly when a program calls a standard procedure such as `sqrt()`, and they use libraries implicitly for I/O, conversions, sorting, and many other functions too complex to express as inline code. As languages have gotten more complex, libraries have gotten correspondingly more complex. When I wrote a Fortran-77 compiler 20 years ago, the run-time library was already more work than the compiler itself, and a Fortran-77 library is far simpler than one for C++.

The growth of language libraries means not only that all programs include library code but that most programs include a lot of the same library code. Every C program, for example, uses the system call library, nearly all use the standard I/O library routines such as `printf`, and many use other popular libraries for math, networking, and other common functions. This means that in a typical UNIX system with a thousand compiled programs, there are close to a thousand copies of `printf`. If all those programs could share a single copy

of the library routines they use, the savings in disk space would be substantial. (On a UNIX system without shared libraries, there are 5–10MB of copies of `printf` alone.) Even more important, if running programs could share a single in-memory copy of the libraries, the main memory savings could be very significant, both in saving memory and in improving paging behavior.

All shared-library schemes work essentially the same way. At link time, the linker searches through libraries as usual to find modules that resolve otherwise undefined external symbols. But rather than copying the contents of the module into the output file, the linker makes a note of what library the module came from and puts a list of the libraries in the executable. When the program is loaded, startup code finds those libraries and maps them into the program's address space before the program starts (Figure 9.1). Standard operating system file-mapping semantics automatically share pages that are mapped read-only or copy-on-write. The startup code that does the mapping may be in the operating system, in the executable, in a special dynamic linker mapped into the process's address space, or in some combination of the three.

In this chapter, we look at statically linked shared libraries, that is, libraries where program and data addresses in libraries are bound to executables at link time. In Chapter 10 we look at the considerably more complex dynamically linked libraries. Although dynamic linking is more flexible and more modern, it's also a lot slower than static linking because a great deal of work that would otherwise have been done once at link time is redone each time a dynamically linked program starts. Also, dynamically linked programs usually use extra glue code to call routines in shared libraries. The glue usually contains several jumps, which can slow down calls considerably. On systems that support both static and dynamic shared libraries, unless programs need the extra flexibility of dynamic linking, they're faster and smaller with statically linked libraries.

9.1 BINDING TIME

Shared libraries raise binding-time issues that don't apply to conventionally linked programs. A program that uses a shared library depends on having that shared library available when the program is run. One kind of error occurs when the required libraries aren't present. There's not much to be done in that case other than printing a cryptic error message and exiting.

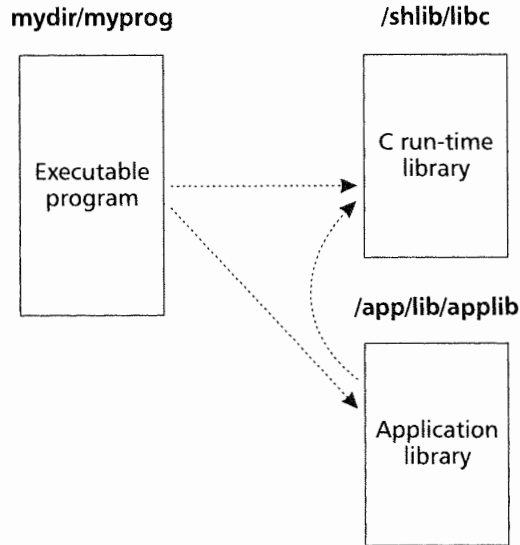


FIGURE 9.1 • Program with shared libraries.

A much more interesting problem occurs when the library is present, but the library has changed since the program was linked. In a conventionally linked program, symbols are bound to addresses and library code is bound to the executable at link time, so the library the program was linked with is the one it uses regardless of subsequent changes to the library. With static shared libraries, symbols are still bound to addresses at link time, but library code isn't bound to the executable until run time. (With dynamic shared libraries, they're both delayed until run time.)

A static shared library can't change very much without breaking the programs that it is bound to. Because the addresses of routines and data in the library are bound into the program, any changes in the addresses to which the program is bound will cause havoc.

A static shared library can sometimes be updated without breaking the programs that use it, if the updates can be made in a way that doesn't move any addresses in the library that programs depend on. This permits minor version updates, typically for small bug fixes. Larger changes unavoidably change program addresses, which means that a system either needs multiple versions of the library or forces programmers to relink all their programs each

time the library changes. In practice, the solution is invariably multiple versions, since disk space is cheap and tracking down every executable that might have used a shared library is rarely possible.

9.2 SHARED LIBRARIES IN PRACTICE

In the rest of this chapter we concentrate on the static shared libraries provided in UNIX System V Release 3.2 (COFF format), older Linux systems (a.out format), and the BSD/OS derivative of 4.4BSD (a.out and ELF formats). All three work nearly the same, but some of the differences are instructive. The System V Release 3.2 implementation required changes in the linker to support searching shared libraries and extensive operating system support to do the run-time startup required. The Linux implementation required one small tweak to the linker and added a single system call to assist in library mapping. The BSD/OS implementation made no changes at all to the linker or operating system, using a shell script to provide the necessary arguments to the linker and a modified version of the standard C library startup routine to map in the libraries.

9.3 ADDRESS SPACE MANAGEMENT

The most difficult aspect of static shared libraries is address space management. Each shared library occupies a fixed piece of address space in each program in which it is used. Different libraries have to use non-overlapping addresses if they can be used in the same program. Although it's possible to check mechanically that libraries don't overlap, assigning address space to libraries is a black art. On the one hand, you want to leave some space in between them so that if a new version of one library grows a little, it won't bump into the next library up. On the other hand, you'd like to put your popular libraries as close together as possible to minimize the number of page tables needed. (Recall that on an x86, for example, there's a second-level table for each 4MB block of address space that is active in a process.)

There's invariably a master table of shared-library address space on each system, with libraries starting someplace in the address space far away from applications. Linux's start at hex 60000000, BSD/OS's at a0000000. Commercial vendors subdivide the address space further between vendor-supplied libraries and user and third-party libraries (which start at a0800000 in BSD/OS, for example).

Generally both the code and data addresses for each library are explicitly defined, with the data area starting on a page boundary a page or two after the end of the code. This makes it possible to create minor version updates, because the updates frequently don't change the data layout, but just add or change code.

Each individual shared library exports symbols, both code and data, and usually also imports symbols if the library depends on other libraries. Although it would work if we just linked routines together into a shared library in a haphazard order, real libraries use some discipline in assigning addresses to make it easier—or at least possible—to update a library without changing the addresses of exported symbols. For code addresses, rather than exporting the actual address of each routine, the library contains a table of jump instructions that jump to all of the routines, with the addresses of the jump instructions exported as the addresses of the routines. All jump instructions are the same size, so the addresses in the jump table are easy to compute and won't change from version to version so long as no entries are added or deleted in the middle of the table. One extra jump per routine is an insignificant slowdown, but because the actual routine addresses are not visible, new versions of the library will be compatible even if routines in the new version aren't all the same sizes and addresses as in the old version.

For exported data, the situation is more difficult, because there's no easy way to add a level of indirection like the one for code addresses. In practice, it turns out that exported data tend to be tables of known size that change rarely, such as the array of `FILE` structures for the C standard I/O library or single word values like `errno` (the error code from the most recent system call) or `tzname` (pointers to two strings giving the name of the local time zone). With some manual effort, the programmer who creates the shared library can collect the exported data at the front of the data section to precede any anonymous data that are part of individual routines, making it less likely that exported addresses will change from one version to the next.

9.4 STRUCTURE OF SHARED LIBRARIES

The shared library is an executable-format file that contains all of the library code and data, ready to be mapped in (Figure 9.2). Some shared libraries start with a small bootstrap routine that is used to map in the rest of the library. After that comes the jump table, aligned on a page boundary if it's not the first thing in the library. The exported address of each public routine in the library is the jump table entry. Following the jump table is the rest of the text section (the jump table is considered to be text, because it's executable code), then the exported data and private data. The bss segment logically follows the data, but as in any other executable file, isn't actually present in the file.

9.5 CREATING SHARED LIBRARIES

A UNIX shared library actually consists of two related files, the shared library itself and a stub library for the linker to use. A library creation utility takes as input a normal library in archive format and some files of control information and uses them to create the two files. The stub library contains no code or data at all (other than possibly a tiny bootstrap routine) but contains symbol definitions for programs linked with the library to use.

Creating the shared library involves the following basic steps, which we discuss in greater detail below:

1. Determine at what address the library's code and data will be loaded.
2. Scan through the input library to find all of the exported code symbols. (One of the control files may be a list of some symbols not to export, if they're just used for inter-routine communication within the library.)
3. Make up the jump table with an entry for each exported code symbol.
4. If there's an initialization or loader routine at the beginning of the library, compile or assemble that.
5. Create the shared library: Run the linker and link everything together into one big executable-format file.

File header, (a.out/COFF/ELF)
(Initialization routine)
Jump table
Code
Global data
Private data

FIGURE 9.2 • Structure of typical shared library.

6. Create the stub library: Extract the necessary symbols from the newly created shared library, reconcile those symbols with the symbols from the input library, create a stub routine for each library routine, then compile or assemble the stubs and combine them into the stub library. (In COFF libraries, there's also a little initialization code placed in the stub library to be linked into each executable.)

CREATING THE JUMP TABLE

The easiest way to create the jump table is to write an assembler source file full of jump instructions (Figure 9.3) and assemble it. Each jump instruction needs to be labeled in a systematic way so that the addresses can later be extracted for the stub library.

A minor complication occurs on architectures such as the x86 that have different sizes of jump instructions. For libraries containing less than 64KB of code, short 3-byte jumps are adequate. For libraries larger than that, longer 5-byte jumps are necessary. Mixed sizes of jumps aren't very satisfactory, both because it makes the table addresses harder to compute and because it makes it far harder to make the jump table compatible in future builds of the library.

```

... start on a page boundary

        .align 8 ; align on 8-byte boundary for variable length insns
JUMP_read: jmp _read
        .align 8
JUMP_write: jmp _write

later in library

_read: ... code for read()
...
_write: ... code for write()

```

FIGURE 9.3 • Jump table.

The simplest solution is to make all of the jumps the largest size. Alternatively, we can make all of the jumps short, and for routines that are too far away for short jumps, we can generate anonymous long jump instructions at the end of the table to which short instructions can jump. (That's usually more trouble than it's worth, because jump tables are rarely more than a few hundred entries in the first place.)

CREATING THE SHARED LIBRARY

Once the jump table and, if needed, the loader routine are created, the creation of the shared library is easy. Just run the linker with suitable switches to make the code and data start at the right places and link together the bootstrap, the jump tables, and all of the routines from the input library. This both assigns addresses to everything in the library and creates the shared-library file.

One minor complication involves interlibrary references. If you're creating, say, a shared math library that uses routines from the shared C library, the references have to be made correctly. Assuming that the library whose routines are needed has already been built when the linker builds the new library, the linker need only search the old library's stub library, just like any normal executable that refers to the old library. This will make all of the references

correct. The only remaining issue is that there needs to be some way to ensure that any programs that use the new library also link to the old library. Suitable design of the new stub library can ensure that.

CREATING THE STUB LIBRARY

Creating the stub library is one of the trickier parts of the shared-library process. For each routine in the real library, the stub library needs to contain a corresponding entry that defines both the exported and the imported global symbols.

The global data symbols are wherever the linker puts them in the shared-library image, and the most reasonable way to get their values is to create the shared library with a symbol table and to extract the symbols from that symbol table. For global code symbols, the entry points are all in the jump table, so it's equally easy to extract the symbols from the shared library or to compute the addresses from the base address of the jump table and each symbol's position in the table.

Unlike a normal library module, a module in the stub library contains no code or data, just symbol definitions. The symbols have to be defined as absolute rather than relocatable numbers, because the shared library has already had all of its relocation done. The library creation program extracts each routine from the input library, and from that routine it gets the defined and undefined globals as well as the type (text or data) of each global. It then writes the stub routine—usually as a little assembler program—defining each text global as the address of the jump table entry, each data or bss global as the actual address in the shared library, and each undefined global as undefined. When it has a complete set of stub sources, it assembles them all and combines them into a normal library archive.

COFF stub libraries use a different, more primitive design. They're single object files with two named sections. The `.lib` section contains all of the relocation information pointing at the shared library, and the `.init` section contains initialization code that is linked into each client program, typically to initialize variables in the shared library.

Linux shared libraries are simpler still. They use an `a.out` file containing the symbol definitions with set vector symbols (described in Section 9.6) for use at program link time.

VERSION NAMING

Shared libraries in UNIX systems are assigned names that are mechanically derived from the original library, adding a version number. If the original library was called `/lib/libc.a` (the usual name for the C library) and the current library version is 4.0, then the stub library might be `/lib/libc_s.4.0.0.a` and the shared-library image might be `/shlib/libc_s.4.0.0`. (The extra zero allows for minor version updates.) Once the libraries are moved into the appropriate directories they're ready to use.

Any shared-library system needs a way to handle multiple versions of libraries. When a library is updated, the new version may or may not be address compatible and call compatible with previous versions. (UNIX systems address this issue with the multinumber version names mentioned above.)

Generally, the first number changes each time a new incompatible version of the library is released. A program linked with a 4.x.x library can't use a 3.x.x or a 5.x.x library. The second number is the minor version. On Sun systems, each executable requires a minor version at least as great as the one with which the executable was linked. If it were linked with 4.2.x, for example, it would run with a 4.3.x library but not a 4.1.x. Other systems treat the second component as an extension of the first component, so an executable linked with a 4.2.x library will only run with a 4.2.x library. The third component is universally treated as a patch level. Executables prefer the highest available patch level, but any patch level will do.

Different systems take slightly different approaches to finding the appropriate libraries at run time. Sun systems have a fairly complex run-time loader that looks at all of the file names in the library directory and picks the best one. Linux systems use symbolic links to avoid the search process. If the latest version of the `libc.so` library is version 4.2.2, the library's name is `libc_s.4.2.2`, but the library is also linked to `libc_s.4.2`, so the loader need only open the shorter name and the correct version is selected.

Most systems permit shared libraries to reside in multiple directories. An environment variable such as `LD_LIBRARY_PATH` can override the path built into the executable, permitting developers to substitute library versions in their private directories for debugging or performance testing. (Programs that use the "set user ID" feature to run as other than the current user have to ignore `LD_LIBRARY_PATH` to prevent a malicious user from substituting a Trojan horse library.)

9.6 LINKING WITH SHARED LIBRARIES

Linking with static shared libraries is far simpler than creating the libraries, because the process of creating the stub libraries has already done nearly all the hard work to make the linker resolve program addresses to the appropriate places in the libraries. The only hard part is arranging for the necessary shared libraries to be mapped in when the program starts.

Each format provides a trick to let the linker create a list of libraries that startup code can use to map in the libraries. COFF libraries use a brute-force approach; ad hoc code in the linker creates a section in the COFF file with the names of the libraries. The Linux linker has a somewhat less brute-force approach that creates a special symbol type called a *set vector*. Set vectors are treated like normal global symbols, except that if there are multiple definitions, the definitions are all put in an array named by the symbol. Each shared-library stub defines a set vector symbol `__SHARED_LIBRARIES__` that is the address of a structure containing the name, version, and load address of the library. The linker creates an array of pointers to each of those structures and calls it `__SHARED_LIBRARIES__` so that the run-time startup code can use it.

The BSD/OS shared-library scheme creates the list externally and passes it to the regular linker. The shell script wrapper used to create a shared executable runs down the list of libraries that are passed as arguments to the command or that are used implicitly (the C library), extracts the file names and load addresses for those libraries from a list in a system file, writes a little assembler source file containing an array of structures that contain library names and load addresses, assembles that file, and includes the object file in the list of arguments to the linker.

In each case, the references from the program code to the library addresses are resolved automatically from the addresses in the stub library.

9.7 RUNNING WITH SHARED LIBRARIES

Starting a program that uses shared libraries involves three steps: loading the executable, mapping the libraries, and doing library-specific initialization. In each case, the program executable is loaded into memory by the system in the

usual way. After that, the different schemes diverge. The System V.3 kernel had extensions to handle COFF shared-library executables, and the kernel internally looked at the list of libraries and mapped them in before starting the program. The disadvantages of this scheme were “kernel bloat” (adding more code to the nonpageable kernel) and inflexibility, because it didn’t permit any flexibility or upgradability in future versions. System V.4 scrapped the whole scheme and went to ELF dynamic shared libraries, which we address in Chapter 10.

Linux adds a single `uselib()` system call that takes the file name and address of a library and maps it into the program address space. The startup routine bound into the executable runs down the list of libraries, doing a `uselib()` on each.

The BSD/OS scheme uses the standard `mmap()` system call that maps pages of a file into the address space and a bootstrap routine that is linked into each shared library as the first thing in the library. The startup routine in the executable runs down the table of shared libraries, and for each one opens the file, maps the first page of the file to the load address, and then calls the bootstrap routine, which is at a fixed location near the beginning of that page following the executable file header. The bootstrap routine then maps the rest of the text segment, maps the data segment, and maps fresh address space for the bss segment, then returns.

Once the segments are all mapped, there’s often some library-specific initialization to do (for example, putting a pointer to the system environment strings in the global variable `environ` specified by standard C). The COFF implementation collects the initialization code from the `.init` segments in the program file and runs it from the program startup code. Depending on the library, it may or may not call routines in the shared library. The Linux implementation doesn’t do any library initialization and the documentation notes that variables defined in both the program and the library don’t work very well.

In the BSD/OS implementation, the bootstrap routine for the C library receives a pointer to the table of shared libraries and maps in all of the other libraries, minimizing the amount of code that has to be linked into individual executables. Recent versions of BSD use ELF format executables. The ELF header has an `interp` section that contains the name of an interpreter program to use when running the file. BSD uses the shared C library as the interpreter, which means that the kernel maps in the shared C library before the

program starts, saving the overhead of some system calls. The library bootstrap routine does the same initializations, maps the rest of the libraries and, through a pointer, calls the main routine in the program.

9.8 THE malloc HACK AND OTHER SHARED-LIBRARY PROBLEMS

Although static shared libraries have excellent performance, their long-term maintenance is difficult and error prone. This section gives an anecdote that illustrates this difficulty and also mentions other additional problems that occur in shared libraries.

In a static library, all intralibrary calls are permanently bound, and it's not possible to substitute a private version of a routine by redefining the routine in a program that uses the library. For the most part, that's not a problem because few programs redefine standard library routines like `read()` or `strcmp()`, or even if they do it's not a major problem if the program uses a private version of `strcmp()` while routines in the library call the standard version.

But a lot of programs define their own versions of `malloc()` and `free()`, the routines that allocate heap storage, and multiple versions of those routines in a program don't work. The standard `strdup()` routine, for example, returns a pointer to a string allocated by `malloc`, which the application can free up when no longer needed. If the library allocated the string with one version of `malloc` but the application freed that string with a different version of `free`, chaos would ensue.

To permit applications to provide their own versions of `malloc` and `free`, the System V.3 shared C library uses an ugly hack (Figure 9.4). The system's maintainers redefined `malloc` and `free` as indirect calls through pointers (which we'll call `malloc_ptr` and `free_ptr`) bound into the data part of the shared library:

```
extern void *(*malloc_ptr)(size_t);
extern void (*free_ptr)(void *);
#define malloc(s) (*malloc_ptr)(s)
#define free(s) (*free_ptr)(s)
```

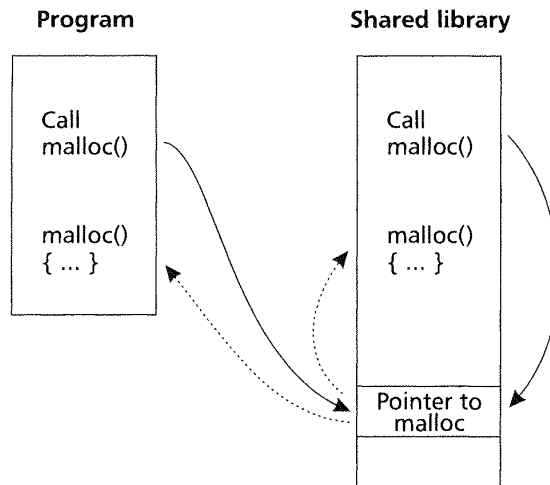


FIGURE 9.4 • The malloc hack.

Then they recompiled the entire C library and added these lines (or the assembler equivalent) to the `.init` section of the stub library, so that they are included in every program that uses the shared library:

```
#undef malloc
#undef free
malloc_ptr = &malloc;
free_ptr = &free;
```

Because the stub library is bound into the application, not the shared library, its references to `malloc` and `free` are resolved at the time each program is linked. If there's a private version of `malloc` and `free`, it puts pointers to them in the pointers; otherwise it will use the standard library version. Either way, the library and the application use the same version of `malloc` and `free`.

Although the implementation of this trick made maintenance of the library harder and doesn't scale to more than a few carefully chosen names, the idea that intralibrary calls can be made through pointers that are resolved at program run time is a good one, so long as it's automated and doesn't

```

#include <stdio.h>

/* extern */
int errno;

main()
{
    unlink("/non-existent-file");
    printf("Status was %d\n", errno);
}

```

FIGURE 9.5 • Address conflict example.

require fragile manual source code tweaks. We'll find out how the automated version works in Chapter 10.

Name conflicts in global data remain a problem with static shared libraries. Consider the small program in Figure 9.5. If you compile and link it with any of the shared libraries we describe in this chapter, it will print a status code of zero rather than the correct error code. That's because `int errno;` defines a new instance of `errno` that isn't bound to the one in the shared library. If you uncomment the `extern`, the program works, because now it's an undefined global reference that the linker binds to the `errno` in the shared library. As we'll see in Chapter 10, dynamic linking solves this problem as well at some cost in performance.

Finally, even the jump table in UNIX shared libraries has been known to cause compatibility problems. From the point of view of routines outside a shared library, the address of each exported routine in the library is the address of the jump table entry. But from the point of view of routines within the library, the address of that routine may be the jump table entry or it may be the real entry point to which the table entry jumps. There have been cases where a library routine examined an address passed as an argument to see if it was one of the other routines in the library, in order to do some special-case processing.

An obvious but less than totally effective solution is to bind the address of the routine to the jump table entry while building the shared library, because that ensures that all symbolic references to routines within the library are

resolved to the table entry. But if two routines are within the same object file, the reference in the object file is usually a relative reference to the routine's address in the text segment. (Because it's in the same object file, the routine's address is known; other than this peculiar case, there's no reason to make a symbolic reference back into the same object file.) Although it would be possible to scan relocatable text references for values that match exported symbol addresses, the most practical solution to this problem is "don't do that," don't write code that depends on recognizing the address of a library routine.

MS Windows DLLs have a similar problem, because within each EXE or DLL the addresses of imported routines are considered to be the addresses of the stub routines that make indirect jumps to the real address of the routine. Again, the most practical solution to the problem is "don't do that."

EXERCISES

EXERCISE • 9.1

If you look in a `/shlib` directory on a UNIX system with shared libraries, you'll usually see three or four versions of each library with names like `libc_s.2.0.1` and `libc_s.3.0.0`. Why not just have the most recent one?

EXERCISE • 9.2

In a stub library, why is it important to include all of the undefined globals for each routine, even if the undefined global refers to another routine in the shared library?

EXERCISE • 9.3

What difference does it make whether a stub library is a single large executable with all of the library's symbols (as in COFF or Linux) or an actual library with separate modules?



Now we will extend the linker to support static shared libraries. This involves several subprojects, first to create the shared libraries and then to link executables with the shared libraries.

In our system, a shared library is merely an object file that is linked at a given address. There can be no relocations and no unresolved symbol references, although references to other shared libraries are acceptable. Stub libraries are normal directory format or file format libraries, with each entry in the library containing the exported (absolute) and imported symbols for the corresponding library member but no text or data. Each stub library has to tell the linker the name of the corresponding shared library. If you use directory format stub libraries, a file called LIBRARY NAME contains lines of text. The first line is the name of the corresponding shared library, and the rest of the lines are the names of other shared libraries upon which this one depends. (The space prevents name collisions with symbols.) If you use file format libraries, the initial line of the library has the extra fields

```
LIBRARY nnnn pppppp fffff ggggg hhhhh ...
```

where fffff is the name of the shared library and the subsequent fields are the names of any other shared libraries on which it depends.

PROJECT • 9.1

Make the linker produce static shared libraries and stub libraries from regular directory or file format libraries. If you haven't already done so, you'll have to add a linker flag to set the base address at which the linker allocates the segments. The input is a regular library and stub libraries for any other shared libraries on which this one depends. The output is an executable-format shared library containing the segments of all of the members of the input library and a stub library with a stub member corresponding to each member of the input library.

PROJECT • 9.2

Extend the linker to create executables using static shared libraries. Project 9.1 already has most of the work for searching stub libraries for symbol resolution, because the way that an executable refers to symbols in a shared library is the same as the way that one shared library refers to another. The linker needs to put the names of the required libraries in the output file, so that the run-time loader knows what to load. Have the linker create a segment called `.lib` that contains the names of the shared libraries as strings with a null byte separating the strings and two null bytes at the end. Create a symbol `_SHARED_LIBRARIES` that refers to the beginning of the `.lib` section, to which code in the startup routine can refer.

DYNAMIC LINKING AND LOADING

D*ynamic linking* defers much of the linking process until a program starts running or sometimes even later. It provides a variety of benefits that are hard to get otherwise:

- Dynamically linked shared libraries are easier to create than statically linked shared libraries.
- Dynamically linked shared libraries are easier to update than statically linked shared libraries.
- The semantics of dynamically linked shared libraries can be much closer to those of unshared libraries.
- Dynamic linking permits a program to load and unload routines at run time, a facility that can otherwise be very difficult to provide.

There are a few disadvantages, of course. The run-time performance costs of dynamic linking are substantial compared to those of static linking, because a large part of the linking process has to be redone every time a program runs. Every dynamically linked symbol used in a program has to be looked up

- in a symbol table and resolved. (MS Windows dynamic-link libraries mitigate
- this cost somewhat, as we describe in Section 10.7.) Dynamic shared libraries
- are also larger than static shared libraries, because the dynamic ones have to
- include symbol tables.

Beyond issues of call compatibility, a chronic source of problems is changes in library semantics. Because dynamic shared libraries are so easy to update compared to unshared or static shared libraries, it's easy to change libraries that are in use by existing programs, which means that the behavior of those programs changes even though "nothing has changed." This is a frequent source of problems on MS Windows, where programs use a lot of shared libraries, libraries go through a lot of versions, and library version control is not very sophisticated. Most programs ship with copies of all of the libraries they use, and installers often will inadvertently install an older version of a shared library on top of a newer one, breaking programs that are expecting features found in the newer one. Well-behaved applications pop up a warning before installing an older library over a newer one, but even so, programs that depend on the semantics of older libraries have been known to break when newer versions replace the older ones.

10.1 ELF DYNAMIC LINKING

The Sun Microsystems SunOS introduced dynamic shared libraries to UNIX in the late 1980s. UNIX System V Release 4, which Sun co-developed, introduced the ELF object format and adapted the Sun scheme to ELF. ELF was clearly an improvement over the previous object formats, and by the late 1990s it had become the standard for UNIX and UNIX-like systems, including Linux and BSD derivatives.

10.2 CONTENTS OF AN ELF FILE

As discussed in Chapter 3, an ELF file can be viewed as a set of *sections*, interpreted by the linker, or a set of *segments*, interpreted by the program loader.

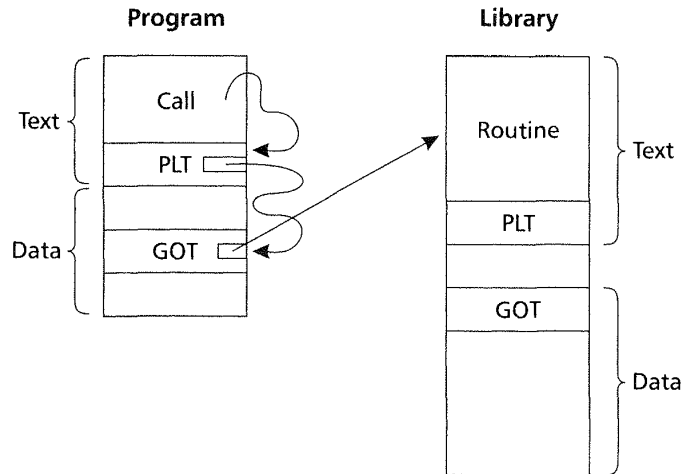


FIGURE 10.1 • PLT and GOT.

ELF programs and shared libraries have the same general structure, but with different sets of segments and sections.

ELF shared libraries can be loaded at any address, so they invariably use position-independent code (PIC) so that the text pages of the file need not be relocated and can be shared among multiple processes. As described in Chapter 8, ELF linkers support PIC code with a global offset table (GOT) in each shared library that contains pointers to all of the static data referenced in the program. The dynamic linker resolves and relocates all of the pointers in the GOT. This can be a performance issue, but in practice the GOT is small except in very large libraries; a commonly used version of the standard C library has only 180 entries in the GOT for over 350KB of code.

Because the GOT is in the same loadable ELF file as the code that references it, and the relative addresses within a file don't change regardless of where the program is loaded, the code can locate the GOT with a relative address, load the address of the GOT into a register, and then load pointers from the GOT whenever it needs to address static data. A library need not have a GOT if it references no static data (but in practice all libraries do).

To support dynamic linking, each ELF shared library and each executable that uses shared libraries has a procedure linkage table, or PLT (Figure 10.1). The PLT adds a level of indirection for function calls analogous to that

provided by the GOT for data. The PLT also permits lazy evaluation, that is, not resolving procedure addresses until they're called for the first time. Because the PLT tends to have a lot more entries than the GOT (over 600 in the C library mentioned above), and because most of the routines will never be called in any given program, lazy evaluation can both speed startup and save considerable time overall. The details of the PLT are discussed in Section 10.4.

An ELF dynamically linked file contains all of the linker information that the run-time linker will need to relocate the file and resolve any undefined symbols. The `.dynsym` section (the dynamic symbol table) contains all of the file's imported and exported symbols. The `.dynstr` and `.hash` sections contain the name strings for the symbol and a hash table the run-time linker can use to look up symbols quickly.

The final extra piece of an ELF dynamically linked file is the DYNAMIC segment (also marked as the `.dynamic` section), which the run-time dynamic linker uses to find the information about the file the linker needs. It's loaded as part of the data segment but is pointed to from the ELF file header so the run-time dynamic linker can find it. The DYNAMIC segment is a list of tagged values and pointers. Some of the following entry types occur just in programs, some just in libraries, some in both:

- **NEEDED.** This is the name of a library this file needs. (Always in programs, sometimes in libraries when one library is dependent on another; can occur more than once.)
- **SONAME** (shared-object name). This is the name of the file the linker uses. (Libraries.)
- **SYMTAB, STRTAB, HASH, SYMENT, STRSZ.** These point to the symbol table, associated string and hash tables, size of a symbol table entry, and size of the string table, respectively. (Both.)
- **PLTGOT.** This points to the GOT or on some architectures to the PLT. (Both.)
- **REL, RELSZ, and RELENT; or RELA, RELASZ, and RELAENT.** These give pointer to, number of, and size of relocation entries, respectively. REL entries don't contain addends; RELA entries do. (Both.)
- **JMPREL, PLTRELSZ, and PLTREL.** These give pointer to, size, and format (REL or RELA) of relocation table, respectively, for data referred to by the PLT. (Both.)

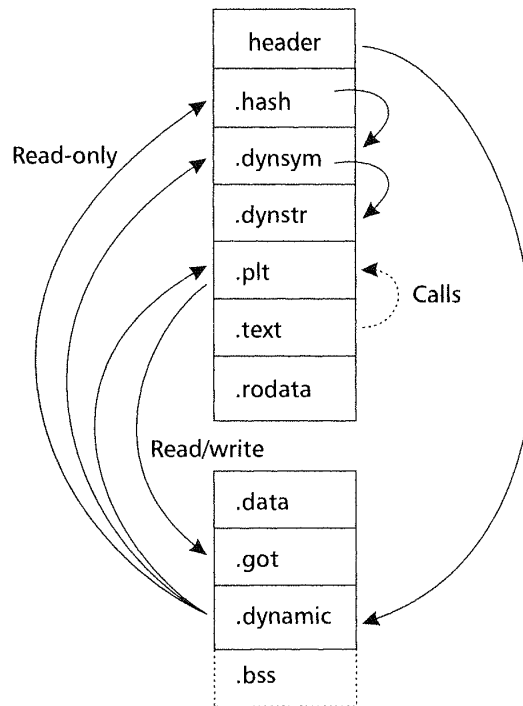


FIGURE 10.2 • An ELF shared library.

- **INIT and FINI.** These point to initializer and finalizer routines to be called at program startup and finish. (Optional but usually in both.)

There are a few other obscure types that are not often used. An entire ELF shared library might look like Figure 10.2. First come the read-only parts, including the symbol table, the PLT, the text, and read-only data; then the read/write parts, including regular data, the GOT, and the dynamic section. The bss logically follows the last read/write section, but as always isn't present in the file.

An ELF program looks much the same, but in the read-only segment has INIT and FINI routines, and an INTERP section near the front of the file to specify the name of the dynamic linker (usually `ld.so`). The data segment has no GOT, because program files aren't relocated at run time.

10.3 LOADING A DYNAMICALLY LINKED PROGRAM

Loading a dynamically linked ELF program is a lengthy but straightforward process.

STARTING THE DYNAMIC LINKER

When the operating system runs the program, it maps in the file's pages as normal but notes that there's an `INTERPRETER` section in the executable. The specified interpreter is the dynamic linker, `ld.so`, which is itself in ELF shared-library format. Rather than starting the program, the system maps the dynamic linker into a convenient part of the address space as well and starts `ld.so`, passing on the stack an *auxiliary vector* of information needed by the linker. The vector includes the following elements:

- `AT_PHDR`, `AT_PHENT`, and `AT_PHNUM`. These give the address of the program header for the program file, the size of each entry in the header, and the number of entries, respectively. This structure describes the segments in the loaded file. If the system hasn't mapped the program into memory, there may instead be an `AT_EXECD` entry that contains the file descriptor on which the program file is open.
- `AT_ENTRY`. This gives the starting address of the program, to which the dynamic linker jumps after it has finished initialization.
- `AT_BASE`. This is the address at which the dynamic linker was loaded.

At this point, bootstrap code at the beginning of `ld.so` finds its own GOT, the first entry of which points to the dynamic segment in the `ld.so` file. From the dynamic segment, the linker can find its own relocation entries, relocate pointers in its own data segment, and resolve code references to the routines needed to load everything else. (The Linux `ld.so` names all of the essential routines with names starting with `_dt_` and special-case code looks for symbols that start with the string and resolves them.)

The linker then initializes a chain of symbol tables with pointers to the program's symbol table and the linker's own symbol table. Conceptually, the program file and all of the libraries loaded into a process share a single symbol table. But rather than build a merged symbol table at run time, the linker

keeps a linked list of the symbol tables in each file. Each file contains a hash table to speed symbol lookup, with a set of hash headers and a hash chain for each header. The linker can search for a symbol quickly by computing the symbol's hash value once, then running through the appropriate hash chain in each of the symbol tables in the list.

FINDING THE LIBRARIES

Once the linker's own initializations are done, it finds the names of the libraries required by the program. The program's program header has a pointer to the dynamic segment that contains dynamic linking information for the program. That segment contains a pointer `DT_STRTAB` to the file's string table and entries `DT_NEEDED`, each of which contains the offset in the string table of the name of a required library.

For each library, the linker finds the library's ELF shared-library file, which is in itself a fairly complex process. The library name in a `DT_NEEDED` entry is something like `libXt.so.6` (the Xt toolkit, version 6). The library file might be in any of several library directories and might not even have the same file name. On my system, the actual name of that library is `/usr/X11R6/lib/libXt.so.6.0`, with the `.0` at the end being a minor version number.

The linker looks in the following places to find the library:

1. If the dynamic segment contains an entry called `DT_RPATH`, it's a colon-separated list of directories to search for libraries. This entry is added by a command line switch or environment variable to the regular (not dynamic) linker at the time a program is linked. It's mostly used for subsystems such as databases that load a collection of programs and supporting libraries into a single directory.
2. If there's an environment symbol `LD_LIBRARY_PATH`, it's treated as a colon-separated list of directories in which the linker looks for the library. This lets a developer build a new version of a library, put it in the `LD_LIBRARY_PATH`, and use it with existing linked programs either to test the new library or equally well to instrument the behavior of the program. (It skips this step if the program is set-uid, for security reasons.)
3. The linker looks in the library cache file `/etc/ld.so.conf`, which contains a list of library names and paths. If the library name is present, it uses the

corresponding path. This is the usual way that most libraries are found. (The file name at the end of the path need not be exactly the same as the library name; see Library Versions in Section 10.5.)

4. If all else fails, the linker looks in the default directory `/usr/lib`; if the library's still not found, it displays an error message and exits.

Once it has found the file containing the library, the dynamic linker opens the file and reads the ELF header to find the program header, which in turn points to the file's segments including the dynamic segment. The linker allocates space for the library's text and data segments and maps them in, along with zeroed pages for the bss. From the library's dynamic segment, it adds the library's symbol table to the chain of symbol tables, and if the library requires further libraries that are not already loaded, it adds any new libraries to the list to be loaded. When this process terminates, all of the libraries have been mapped in, and the loader has a logical global symbol table consisting of the union of all of the symbol tables of the program and the mapped library.

SHARED-LIBRARY INITIALIZATION

Now the loader revisits each library and handles the library's relocation entries, filling in the library's GOT and performing any relocations needed in the library's data segment. Load-time relocations on an x86 include the following:

- `R_386_GLOB_DAT`, used to initialize a GOT entry to the address of a symbol defined in another library.
- `R_386_32`, a non-GOT reference to a symbol defined in another library (generally a pointer in static data).
- `R_386_RELATIVE`, for relocatable data references, typically a pointer to a string or other locally defined static data.
- `R_386_JMP_SLOT`, used to initialize GOT entries for the PLT (see Section 10.4).

If a library has an `.init` section, the loader calls it to do library-specific initializations, such as C++ static constructors, and any `.fini` section is noted

to be run at exit time. (It doesn't do the `.init` for the main program, because that's handled in the program's own startup code.) When this pass is done, all of the libraries are fully loaded and ready to execute, and the loader calls the program's entry point to start the program.

10.4 LAZY PROCEDURE LINKAGE WITH THE PLT

Programs that use shared libraries generally contain calls to a lot of functions. In a single run of the program, many of the functions are never called (in error routines or other parts of the program that aren't used). Furthermore, each shared library also contains calls to functions in other libraries, even fewer of which will be executed in a given program run because many of them are in routines that the program never calls either directly or indirectly.

To speed program startup, dynamically linked ELF programs use lazy binding of procedure addresses; that is, the address of a procedure isn't bound until the first time the procedure is called. This is accomplished by means of a PLT. Each dynamically bound program and shared library has a PLT, with the PLT containing an entry for each nonlocal routine called from the program or library (Figure 10.3). Note that the PLT in PIC code is itself PIC, so it can be part of the read-only text segment.

All calls within the program or library to a particular routine are adjusted (when the program or library is built) to be calls to the routine's entry in the PLT. The first time the program or library calls a routine, the PLT entry calls the run-time linker to resolve the actual address of the routine. After that, the PLT entry jumps directly to the actual address, so after the first call, the cost of using the PLT is a single extra indirect jump at a procedure call and nothing at a return.

The first entry in the PLT, which we call `PLT0`, is special code to call the dynamic linker. At load time, the dynamic linker automatically places two values in the GOT. At `GOT+4` (the second word of the GOT), it puts a code that identifies the particular library. At `GOT+8`, it puts the address of the dynamic linker's symbol resolution routine.

The rest of the entries in the PLT, which we call `PLT $n$` , each start with an indirect jump through a GOT entry. Each PLT entry has a corresponding GOT entry that is initially set to point to the push instruction in the PLT entry that follows the first `jmp`. (In a PIC file, this requires a load-time relocation but not

Special first entry:

```
PLT0: pushl   GOT+4
      jmp     *GOT+8
```

Regular entries, non-PIC code:

```
PLTn: jmp     *GOT+m
      push    #reloc_offset
      jmp     PLT0
```

Regular entries, PIC code:

```
PLTn: jmp     *GOT+m(%ebx)
      push    #reloc_offset
      jmp     PLT0
```

FIGURE 10.3 • PLT structure in x86 code.

an expensive symbol lookup.) Following the jump is a push instruction that pushes a relocation offset, the offset in the file's relocation table of a special relocation entry of type `R_386_JMP_SLOT`. The relocation entry's symbol reference points to the symbol in the file's symbol table, and its address points to the GOT entry.

This compact but rather complex arrangement means that the first time the program or library calls a PLT entry, the first jump in the PLT entry in effect does nothing, because the GOT entry through which it jumps points back into the PLT entry. Then the push instruction pushes the offset value, which indirectly identifies both the symbol to resolve and the GOT entry into which to resolve it, and jumps to PLT0. The instructions in PLT0 push another code that identifies which program or library it is, and then jump into stub code in the dynamic linker with the two identifying codes at the top of the stack. Note that this is a jump, rather than a call; above the two identifying words just pushed is the return address back to the routine that called into the PLT.

Now the stub code saves all the registers and calls an internal routine in the dynamic linker to do the resolution. The two identifying words suffice to find the library's symbol table and the routine's entry in that symbol table. The

Main program:

```
extern int token;
```

Routine in shared library:

```
int token = 42;
```

FIGURE 10.4 • Global data initialization.

dynamic linker looks up the symbol value using the concatenated run-time symbol table and stores the routine's address into the GOT entry. Then the stub code restores the registers, pops the two words that the PLT pushed, and jumps off to the routine. With the GOT entry having been updated, subsequent calls to that PLT entry jump directly to the routine itself without entering the dynamic linker.

10.5 OTHER PECULIARITIES OF DYNAMIC LINKING

The ELF linker and the dynamic linker have a lot of obscure code to handle special cases and to try to keep the run-time semantics as similar as possible to those of unshared libraries.

STATIC INITIALIZATIONS

If a program has an external reference to a global variable that is defined in a shared library, the linker has to create in the program a copy of the variable, because program data addresses have to be bound at link time (Figure 10.4). This poses no problem for the code in the shared library, because the code can refer to the variable by means of a GOT pointer that the dynamic linker can fix up, but there is a problem if the library initializes the variable. To deal with this problem, the linker puts an entry in the program's relocation table—which otherwise just contains `R_386_JMP_SLOT`, `R_386_GLOB_DAT`, `R_386_32`, and `R_386_RELATIVE` entries—of type `R_386_COPY` that points to the place in the

program where the copy of the variable is defined and that tells the dynamic linker to copy the initial value of that word of data from the shared library.

Although this feature is essential for certain kinds of code, it occurs very rarely in practice. This is a band-aid, because it only works for single-word data. The initializers that do occur are always pointers to procedures or other data, so the band-aid suffices.

LIBRARY VERSIONS

Dynamic libraries are generally named with major and minor version numbers such as `libc.so.1.1`, but programs should be bound only to major version numbers such as `libc.so.1` because minor versions are supposed to be upward compatible. To keep program loading reasonably fast, the system manager maintains a cache file containing the full pathname of the most recent version of each library, which is updated by a configuration program whenever a new library is installed.

To support this design, each dynamically linked library can have a “true name” called the SONAME (shared-object name) that is assigned at library creation time. For example, the library called `libc.so.1.1` would have a SONAME of `libc.so.1`. (The SONAME defaults to the library’s name.) When the linker builds a program that uses shared libraries, it lists the SONAMEs of the libraries it used rather than the actual names of the libraries. The cache creation program scans all of the directories that contain shared libraries, finds all of the shared libraries, extracts the SONAME from each one, and where there are multiple libraries with the same SONAME, discards all but the highest version number. Then it writes the cache file with SONAMEs and full pathnames so that at run time the dynamic linker can quickly find the current version of each library.

10.6 DYNAMIC LOADING AT RUN TIME

Although the ELF dynamic linker is usually called implicitly at program load time and from PLT entries, programs can also call it explicitly using `dlopen()` to load a shared library and `dlsym()` to find the address of a symbol, which is usually a procedure to call. Those two routines are actually simple wrappers

that call back into the dynamic linker. When the dynamic linker loads a library by means of `dlopen()`, it does the same relocation and symbol resolution it does on any other library, so the dynamically loaded program can without any special arrangements call back to routines already loaded and refer to global data in the running program.

This permits users to add extra functionality to programs without access to the source code of the programs and without even having to stop and restart the programs. (This is useful when the program is something like a database or a Web server.) Mainframe operating systems have provided access to exit routines like this since at least the early 1960s, albeit without such a convenient interface, and it's long been a way to add great flexibility to packaged applications. It also provides a way for programs to extend themselves; there's no reason a program can't write a routine in C or C++, run the compiler and linker to create a shared library, then dynamically load and run the new code. (Mainframe sort programs have linked and loaded custom inner-loop code for each sort job for decades.)

10.7 MICROSOFT DYNAMIC-LINK LIBRARIES

Microsoft Windows also provides shared libraries, called dynamic-link libraries (DLLs) in a fashion similar to but somewhat simpler than ELF shared libraries. The design of DLLs changed substantially between the 16-bit Windows 3.1 and the 32-bit Windows NT and Windows 95. This discussion addresses only the more modern Win32 libraries. DLLs import procedure addresses using a PLT-like scheme. Although the design of DLLs would make it possible to import data addresses using a GOT-like scheme, in practice they use a simpler scheme that requires explicit program code to dereference imported pointers to shared data.

In MS Windows, both programs and DLLs are portable executable (PE) format files and are intended to be memory mapped into a process. Unlike Windows 3.1, where all applications share a single address space, Win32 gives each application its own address space, and executables and libraries are mapped into each address space where they are used. For read-only code this doesn't make any practical difference, but for data it means that each application using a DLL gets its own copy of the DLL's data. (That's a slight oversimplification, because PE files can mark some sections as shared data with a single

copy being shared among all applications that use the file, but most data are unshared.)

Loading an MS Windows executable and the DLLs is similar to loading a dynamically linked ELF program, although in the MS Windows case the dynamic linker is part of the kernel. First the kernel maps in the executable file, guided by section information in the PE headers. Then it maps in all of the DLLs that the executable refers to, again guided by the PE headers in each DLL.

PE files can contain relocation entries. An executable generally won't contain them and so has to be mapped at the address for which it was linked. DLLs all contain relocation entries and are relocated when they're mapped in if the address space for which they were linked isn't available. (Microsoft calls run-time relocation *rebasng*.)

All PE files—both executables and DLLs—have an entry point, and the loader calls a DLL's entry point when the DLL is loaded, when the DLL is unloaded, and each time a process thread attaches to or detaches from the DLL. (The loader passes an argument to indicate why it's making each call.) This provides a hook for static initializers and destructors analogous to the ELF `.init` and `.fini` sections.

IMPORTED AND EXPORTED SYMBOLS IN PE FILES

PE supports shared libraries with two special sections of the file: `.edata` (since MSVC 4.2, exported symbols are in `.rdata` rather than `.edata`), for exported data, which lists the symbols exported from a file; and `.idata`, which lists the symbols imported into a file. Program files generally have only an `.idata` section, while DLLs always have an `.edata` section and may have an `.idata` section if they use other DLLs. Symbols can be exported either by symbol name or by *ordinal*, a small integer that gives the index of the symbol in the export address table. Linking by ordinals is slightly more efficient because it avoids a symbol lookup but is considerably more error prone because it's up to the person who builds a DLL to ensure that ordinals stay the same from one library version to another. In practice ordinals are usually used to call system services that rarely change and names are used for everything else.

The `.edata` section contains an export directory table that describes the rest of the section, followed by the tables that define the exported symbols (Figure 10.5). The export address table contains the relative virtual address

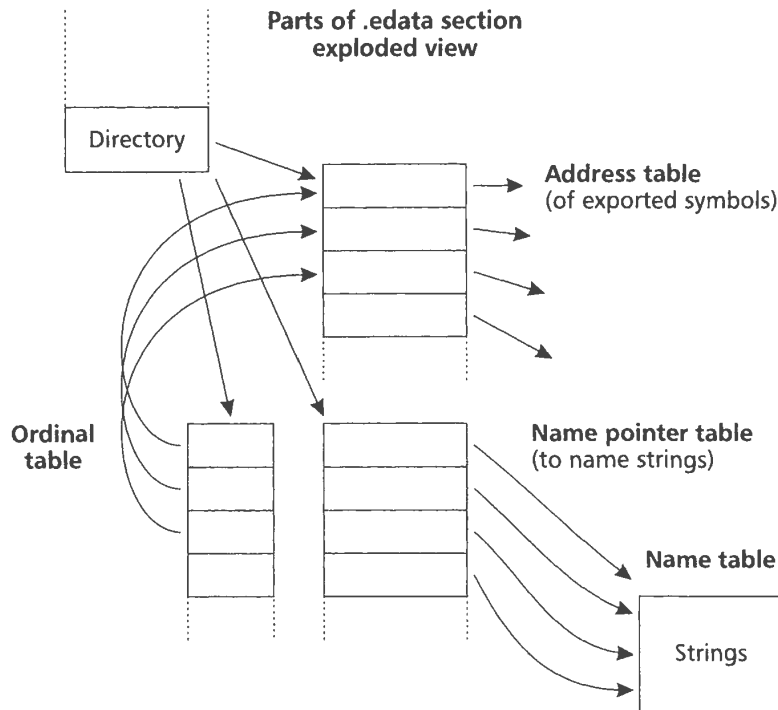


FIGURE 10.5 • Structure of .edata section.

(RVA) of the symbol (relative to the base of the PE file). If the RVA points back into the .edata section, it's a "forwarder" reference, and the value pointed to is a string naming the symbol to use to satisfy the reference (probably defined in a different DLL). The ordinal and name pointer tables are parallel, with each entry in the name pointer table being the RVA of the name string for the symbol and each entry in the ordinal table being the index in the export address table. (Ordinals need not be zero based; the ordinal base to subtract from ordinal values to get the index in the export address table is stored in the export directory and is most often 1.) Exported symbols need not all have names, although in practice they always do. The symbols in the name pointer table are in alphabetical order to permit the loader to use a binary search.

The .idata section does the converse of what the .edata section does, by mapping symbols or ordinals back into virtual addresses. The section consists of a null-terminated array of import directory tables, one per DLL from which

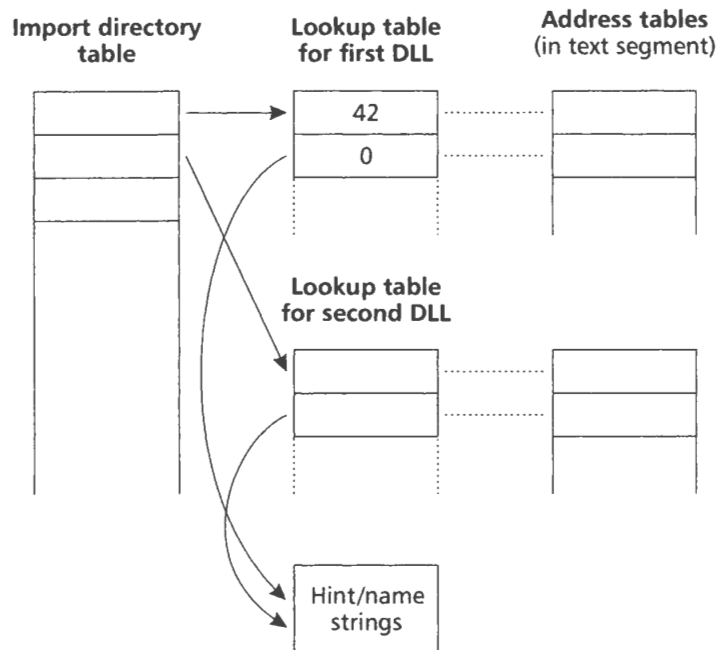


FIGURE 10.6 • Structure of .idata section.

symbols are imported, followed by an import lookup table per DLL, followed by a name table with hints (Figure 10.6).

For each imported DLL, there is an array of import addresses, typically in the program's text segment, into which the program loader places the resolved addresses. The import lookup table identifies the symbols to import, with the entries in the import lookup table being parallel to those in the import address table. The import lookup table consists of 32-bit entries. If the high bit of an entry is set, the low 31 bits are the ordinal of the symbol to import; otherwise the entry is the RVA of an entry in the hint/name table. Each hint/name entry consists of a 4-byte hint that guesses the index of the symbol in the DLL's export name pointer table, followed by the null-terminated symbol name. The program loader uses the hint to probe the export table, and if the symbol name matches, it uses that symbol; otherwise it binary-searches the entire export table for the name. (If the DLL hasn't changed since the program that

uses the DLL was linked, or at least its list of exported symbols hasn't changed, the guess will be right.)

Unlike ELF imported symbols, the values of symbols imported by means of `.idata` are only placed in the import address table; they are not fixed up anywhere else in the importing file. For code addresses, this makes little difference. When the linker builds an executable or DLL, it creates in the text section a table of (misnamed) *thunks*, indirect jumps through the entries in the import address table, and it uses the addresses of the thunks as the address of the imported routine, which is transparent to the programmer. (The thunks as well as most of the data in the `.idata` section actually come from a stub library created at the same time as the DLL.) In recent versions of Microsoft's C and C++ compiler, if the programmer knows that a routine will be called in a DLL, the routine can be declared `__declspec(dllimport)` and the compiler will generate an indirect call to the address table entry, avoiding the extra indirect jump. For data addresses, the situation is more problematic, because it's harder to hide the extra level of indirection required to address a symbol in another executable. Traditionally, programmers just bit the bullet and explicitly declared imported variables to be pointers to the real values and explicitly dereferenced the pointers. Recent versions of Microsoft's C and C++ compiler also let the programmer declare global data to be `__declspec(dllimport)` and the compiler will generate the extra pointer dereferences, much like ELF code that references data indirectly through pointers in the GOT.

LAZY BINDING

Recent versions of MS Windows compilers have added delay-loaded imports to permit lazy symbol binding for procedures, somewhat like the ELF PLT. A delay-loaded DLL contains a structure similar to the `.idata` import directory table, but it is not in the `.idata` section so the program loader doesn't handle it automatically. The entries in the import address table initially all point to a helper routine that finds and loads the DLL and replaces the contents of the address table with the actual addresses. The delay-loaded directory table has a place to store the original contents of the import address table so the values can be put back if the DLL is later unloaded. Microsoft provides a standard helper routine, but its interfaces are documented and programmers can write their own versions if need be. MS Windows also permits programs to load and

unload DLLs explicitly using `LoadLibrary` and `FreeLibrary` and to find addresses of symbols using `GetProcAddress`.

DLLS AND THREADS

One area in which the MS Windows DLL model doesn't work particularly well is thread local storage (TLS). An MS Windows program can start multiple threads in the same process; they share the process's address space. Each thread has a small chunk of TLS to keep data specific to that thread, such as pointers to data structures and resources that the thread is using. The TLS needs slots for the data from the executable and from each DLL that uses TLS. The MS Windows linker can create a `.tls` section in a PE executable that defines the layout for the TLS needed by routines in the executable and any DLLs to which it directly refers. Each time the process creates a thread, the new thread gets its own TLS, created using the `.tls` section as a template.

The problem is that most DLLs can either be linked implicitly from the executable or loaded explicitly with `LoadLibrary`. DLLs loaded explicitly don't automatically get `.tls` storage, and because a DLL's author can't predict whether a library will be invoked implicitly or explicitly, it can't depend on the `.tls` section.

MS Windows defines run-time system calls that allocate slots at the end of the TLS. DLLs use those calls rather than `.tls` unless the DLL is known to only be invoked implicitly.

10.8 OSF/1 PSEUDO-STATIC SHARED LIBRARIES

OSF/1, the ill-fated UNIX variant from the Open Software Foundation, used a shared-library scheme intermediate between static and dynamic linking. Its authors noted that static linking is a lot faster than dynamic linking because less relocation is needed and that libraries are updated infrequently enough that system managers are willing to endure some pain when they update shared libraries (although not the agony of relinking every executable program in the entire system).

So OSF/1 took the approach of maintaining a global symbol table visible to all processes, and it loaded all the shared libraries into a sharable address

space at system boot time. This assigned all of the libraries addresses that wouldn't change while the system was running. Each time a program started, if it used shared libraries it would map in the shared libraries and symbol table and resolve undefined references in the executable using the global symbol table. No load-time relocation was ever required because programs were all linked to load in a part of the address space that was guaranteed to be available in each process, and the library relocation had already happened when they were loaded at boot time. When one of the shared libraries changed, the system just had to be rebooted normally, at which point the system loaded the new libraries and created a new symbol table for executables to use.

This scheme was clever, but it wasn't very satisfactory. For one thing, processing symbol lookups is considerably slower than processing relocation entries, so avoiding relocation wasn't that much of a performance advantage. For another, dynamic linking provides the ability to load and run a library at run time, and the OSF/1 scheme didn't provide for that.

10.9 MAKING SHARED LIBRARIES FAST

Shared libraries—and ELF shared libraries in particular—can be very slow. The slowdowns come from a variety of sources, several of which we mentioned in Chapter 8:

- Load-time relocation of libraries
- Load-time symbol resolution in libraries and executables
- Overhead resulting from PIC function prologue code
- Overhead resulting from PIC indirect data references
- Slower code resulting from PIC reserved addressing registers

The first two problems can be ameliorated by cacheing, the latter three by retreating from pure PIC code.

On modern computers with large address spaces, it's usually possible to choose an address range for a shared library that's available in all or at least most of the processes that use the library. One very effective technique is similar to the MS Windows approach. Either when the library is linked or the first

time a library is loaded, tentatively bind its addresses to a block of address space. After that, each time a program links to the library, use the same addresses if possible (which means that no relocation will be necessary). If that address space isn't available in a new process, the library is relocated as before.

SGI systems use the term *quickstart* to describe the process of pre-relocating objects at link time or in a separate pass over the shared library. BeOS caches the relocated library the first time it's loaded into a process. If multiple libraries depend on each other, in principle it should be possible to pre-relocate and then pre-resolve symbol references among libraries, although I'm not aware of any linkers that do so.

If a system uses pre-relocated libraries, PIC becomes a lot less important. All the processes that load a library at its pre-relocated address can share the library's code whether it's PIC or not, so a non-PIC library at a well-chosen address can in practice be as sharable as PIC without the performance loss of PIC. This is basically the statically linked shared library approach from Chapter 9, except that in the case of address space collisions, rather than the program failing, the dynamic linker moves the libraries at some loss of performance. MS Windows uses this approach.

BeOS implements cached relocated libraries with great thoroughness, including preserving correct semantics when libraries change. When a new version of a library is installed, BeOS notes the fact and creates a new cached version rather than using the old cached version when programs refer to the library. Library changes can have a ripple effect. When library A refers to symbols in library B and library B is updated, a new cached version of library A will also have to be created if any of the referenced symbols in library B have moved. This does make the programmer's life easier, but it's not clear to me that libraries are in practice updated often enough to merit the considerable amount of system code needed to track library updates.

10.10 COMPARISON OF DYNAMIC LINKING APPROACHES

The UNIX/ELF and MS Windows/PE dynamic linking methods differ in several interesting ways. The ELF scheme uses a single name space per program,

while the PE scheme uses a name space per library. An ELF executable lists the symbols it needs and the libraries it needs, but it doesn't record which symbol is in which library. A PE file, on the other hand, lists the symbols to import from each library. The PE scheme is less flexible but also more resistant to inadvertent spoofing: Imagine that an executable calls routine AFUNC (found in library A) and BFUNC (found in library B). If a new version of library A happens to define its own BFUNC, an ELF program could use the new BFUNC in preference to the old one, while a PE program wouldn't. This is a problem with some large libraries; one partial solution is to use the poorly documented DT_FILTER and DT_AUXILIARY fields to tell the dynamic linker which libraries this library imports symbols from, so the linker will search those libraries for imported symbols before searching the executable and the rest of the libraries. The DT_SYMBOLIC field tells the dynamic linker to search the library's own symbol table first, so that other libraries cannot shadow intralibrary references. (This isn't always desirable; consider the malloc hack described in Chapter 9.) These ad hoc approaches make it less likely that symbols in unrelated libraries will inadvertently shadow the correct symbols, but they're no substitute for a hierarchical link-time name space as Java has (Chapter 11).

The ELF scheme tries considerably harder than the PE scheme to maintain the semantics of static linked programs. In an ELF program, references to data imported from another library are automatically resolved, while a PE program needs to treat imported data specially. The PE scheme has trouble comparing the values of pointers to functions, because the address of an imported function is the address of the thunk that calls it, not the address of the actual function in the other library. ELF handles all pointers the same.

At run time, nearly all of the MS Windows dynamic linker is in the operating system, while the ELF dynamic linker runs entirely as part of the application, with the kernel merely mapping in the initial files. The MS Windows scheme is arguably faster, because it doesn't have to map and relocate the dynamic linker in each process before it starts linking. The ELF scheme is definitely a lot more flexible. Because each executable names the interpreter program (now always the dynamic linker, named ld.so) to use, different executables can use different interpreters without requiring any operating system changes. In practice, this makes it easier to support executables from variant versions of UNIX—notably Linux and BSD—by making a dynamic linker that links to compatibility libraries that support nonnative executables.

EXERCISES

EXERCISE • 10.1

ELF shared libraries are often linked so that calls from one routine to another within a single shared library go through the PLT and have their addresses bound at run time. Is this useful? Why or why not?

EXERCISE • 10.2

Imagine that a program calls a library routine `plugh()` that is found in a shared library, and the programmer builds a dynamically linked program that uses that library. Later, the system manager notices that `plugh` is a silly name for a routine and installs a new version of the library that names the routine `xsazq` instead. What happens the next time the programmer runs the program?

EXERCISE • 10.3

If the run-time environment variable `LD_BIND_NOW` is set, the ELF dynamic loader binds all of the program's PLT entries at load time. What would happen in the situation in Exercise 10.2 if `LD_BIND_NOW` were set?

EXERCISE • 10.4

Microsoft implemented lazy procedure binding without operating system assistance by adding some extra cleverness in the linker and by using the existing facilities in the operating system. How hard would it be to provide transparent access to shared data, avoiding the extra level of explicit pointers that the current scheme uses?

WEB

PROJECT

It's impractical to build an entire dynamic linking system for our project linker, because much of the work of dynamic linking happens at run time rather than

link time. Much of the work of building a shared library was already done in Project 8.3, in which we created PIC executables. A dynamically linked shared library is just a PIC executable with a well-defined list of imported and exported symbols and a list of other libraries on which it depends. To mark the file as a shared library or an executable that uses shared libraries, the first line is

```
LINKLIB lib1 lib2 ...
```

or

```
LINK lib1 lib2 ...
```

the `libs` are the names of other shared libraries on which this one depends.

PROJECT • 10.1

Starting with the version of the linker from Project 8.3, extend the linker to produce shared libraries and executables that need shared libraries. The linker needs to take as its input a list of input files to combine into the output executable or library as well as other shared libraries to search. The output file contains a symbol table with defined (exported) and undefined (imported) symbols. Relocation types are the ones for PIC files along with AS4 and RS4 for references to imported symbols.

PROJECT • 10.2

Write a run-time binder, that is, a program that takes an executable that uses shared libraries and resolves its references. It should read in the executable, then read in the necessary libraries, relocate them to non-overlapping available addresses, and create a logically merged symbol table. (You may want to actually create such a table, or you may use a list of per-file tables as ELF does.) Then resolve all of the relocations and external references. When you're done, all code and data should be assigned memory addresses, and all addresses in the code and data should be resolved and relocated to the assigned addresses.